


GEN<math>i</math>CAM		 emva
Version 1.6	GenTL Standard	

GenICam GenTL Standard

Version 1.6



Contents

1	Introduction	12
1.1	Purpose	12
1.2	GenTL Subcommittee	12
1.3	Acronyms and Definitions	12
1.3.1	Acronyms	12
1.3.2	Definitions	13
1.4	References	13
2	Architecture	14
2.1	Overview	14
2.1.1	GenICam GenTL	14
2.1.2	GenICam GenApi	14
2.1.3	GenICam GenTL SFNC	15
2.2	GenTL Modules	15
2.2.1	System Module	16
2.2.2	Interface Module	16
2.2.3	Device Module	17
2.2.4	Data Stream Module	17
2.2.5	Buffer Module	17
2.3	GenTL Module Common Parts	17
2.3.1	C Interface	18
2.3.2	Configuration	19
2.3.3	Signaling (Events)	19
3	Module Enumeration and Instantiation	20
3.1	Setup	20
3.2	System	21
3.3	Interface	22
3.4	Device	24
3.5	Data Stream	25
3.6	Buffer	25
3.7	Enumerated modules list overview	26

3.8	Example.....	27
3.8.1	Basic Device Access	27
3.8.2	InitLib.....	28
3.8.3	OpenTL	28
3.8.4	OpenFirstInterface.....	28
3.8.5	OpenFirstDevice.....	29
3.8.6	OpenFirstDataStream	29
3.8.7	CloseDataStream.....	29
3.8.8	CloseDevice	30
3.8.9	CloseInterface.....	30
3.8.10	CloseTL.....	30
3.8.11	CloseLib	30
4	Configuration and Signaling	31
4.1	Configuration	31
4.1.1	Modules.....	31
4.1.2	XML Description	32
4.1.3	Example.....	35
4.2	Signaling.....	35
4.2.1	Event Objects	36
4.2.2	Event Data Queue.....	38
4.2.3	Event Handling.....	38
4.2.4	Example.....	41
5	Acquisition Engine.....	42
5.1	Overview	42
5.1.1	Announced Buffer Pool.....	42
5.1.2	Input Buffer Pool.....	42
5.1.3	Output Buffer Queue	42
5.2	Acquisition Chain.....	43
5.2.1	Allocate Memory.....	44
5.2.2	Announce Buffers	45
5.2.3	Queue Buffers	46
5.2.4	Register New Buffer Event	46

5.2.5	Start Acquisition.....	46
5.2.6	Acquire Image Data	47
5.2.7	Stop Acquisition.....	47
5.2.8	Flush Buffer Pools and Queues	47
5.2.9	Revoke Buffers.....	47
5.2.10	Free Memory	48
5.3	Buffer Handling Modes.....	48
5.3.1	Default Mode.....	48
5.4	Chunk Data Handling.....	49
5.4.1	Overview	49
5.4.2	Example.....	50
5.5	Data Payload Delivery	51
5.6	Multi-Part Buffer Handling.....	52
5.6.1	Overview	52
5.6.2	Planar Pixel Formats	53
5.6.3	Multiple AOI's.....	53
5.6.4	Pixel Confidence Data.....	53
5.6.5	3D Data Exchange.....	54
5.6.6	Non-Line Oriented Data.....	54
5.6.7	Multi-Source Devices.....	54
5.7	Structured Data Acquisition	55
5.7.1	Composite (Non-contiguous) Buffers	55
5.7.2	Data Stream Flows	55
5.7.3	GenDC Streaming	57
6	Software Interface	59
6.1	Overview	59
6.1.1	Installation.....	59
6.1.2	Function Naming Convention	59
6.1.3	Memory and Object Management.....	60
6.1.4	Thread and Multiprocess Safety.....	60
6.1.5	Error Handling.....	61
6.1.6	Software Interface Versions	63

6.2	Used Data Types	64
6.3	Function Declarations	65
6.3.1	Library Functions	65
6.3.2	System Functions	68
6.3.3	Interface Functions	75
6.3.4	Device Functions	82
6.3.5	Data Stream Functions	89
6.3.6	Port Functions	113
6.3.7	Signaling Functions	123
6.4	Enumerations	129
6.4.1	Library and System Enumerations	129
6.4.2	Interface Enumerations	132
6.4.3	Device Enumerations	133
6.4.4	Data Stream Enumerations	136
6.4.5	Port Enumerations	165
6.4.6	Signaling Enumerations	170
6.5	Structures	174
6.5.1	Data Stream Structures	174
6.5.2	Signaling Structures	177
6.5.3	Port Structures	178
6.6	String Constants	178
6.6.1	Transport Layer Types	178
6.7	Numeric Constants	179
7	Standard Features Naming Convention for GenTL	180
7.1	Common	180
7.1.1	System Module	180
7.1.2	Interface Module	181
7.1.3	Device Module	183
7.1.4	Data Stream Module	184
7.1.5	Buffer Module	185

Figures

Figure 2-1: GenTL Consumer and GenTL Producer	14
Figure 2-2: GenTL Module hierarchy	16
Figure 2-3: GenICam GenTL interface (C and GenApi Feature-interface).....	18
Figure 3-4: Enumeration hierarchy of a GenTL Producer	20
Figure 5-5: Acquisition chain seen from a buffer's perspective	44
Figure 5-6: Default acquisition from the GenTL Consumer's perspective.....	49

Tables

Table 4-1: Local URL definition for XML description files in the module register map.....	33
Table 4-2: Event types per module	36
Table 6-3: Function naming convention	59
Table 6-4: C interface error codes	61
Table 7-5: System module information features	180
Table 7-6: Interface enumeration features	181
Table 7-7: Interface information features.....	181
Table 7-8: Device enumeration features	182
Table 7-9: Device information features	183
Table 7-10: Stream enumeration features	184
Table 7-11: Data Stream information features	184
Table 7-12: Buffer information features	185

Changes

Version	Date	Author	Description
0.1	May 1 st 2007	Rupert Stelz, STEMMER IMAGING	1 st Version
0.2	July 18 th 2007	Rupert Stelz, STEMMER IMAGING	<ul style="list-style-type: none"> • Added Enums • Added Std Features • Added AcqMode Drawings
0.3	November 2007	GenTL Subcommittee: Rupert Stelz, STEMMER IMAGING Sascha Dorenbeck, STEMMER IMAGING Jan Becvar, Leutron Vision Carsten Bienek, IDS Francois Gobeil, Pleora Technologies Christoph Zierl, MVTec	<ul style="list-style-type: none"> • Applied changes as discussed on the last meeting in Ottawa
0.4	January 2008	GenTL Subcommittee	<ul style="list-style-type: none"> • Removed EventGetDataEx and CustomEvent functionality • Added comments from IDS, Matrix Vision, Matrox, Pleora, Leutron Vision, STEMMER IMAGING
1.0	August 2008	Standard Document Release	
1.1	September 2009	GenTL Subcommittee	Changes for V. 1.1 <ul style="list-style-type: none"> • Support of multiple XML-files (Manifest) • Added stacked register access • Changes for using the new endianness scheme • Changes to the installation procedure/location • Added new error codes • Definition of the symbol exports under 64Bit Operating System • Clarifications to the text
1.2	April 2010	GenTL Subcommittee	Changes for V. 1.2 <ul style="list-style-type: none"> • Various clarifications, in particular event objects, feature change handling, event buffer handling • Extension to the BUFFER_INFO_CMD • New error code GC_ERR_NOT_AVAILABLE • Added data payload delivery chapter • Added payload datatype
1.3	August 2011	GenTL Subcommittee	Changes for V. 1.3 <ul style="list-style-type: none"> • Renamed “Acquisition Mode” to “Buffer Handling Mode” • Various clarifications, in particular buffer alignment, error codes, thread safety,

			<p>multiprocess access, default buffer handling mode</p> <ul style="list-style-type: none"> • Added Chunk Data handling in text and function interface • Adjusted Data Stream features to SFNC • Added “Software Interface Version” chapter • Added ptrdiff_t type • New error code GC_ERR_INVALID_ADDRESS • Deprecation of StreamAcquisitionModeSelector, introducing StreamBufferHandlingMode instead. • Clarified buffer alignment
1.4	March 2013	GenTL Subcommittee	<p>Changes for V. 1.4</p> <ul style="list-style-type: none"> • New PAYLOADTYPE_IDs including the ones necessary to reflect GEV2.0 types. Adjusted Chunk-Payloadtypes. • Typos • Clarifications • Removed technology specific names from chapter 7 and referred to GenTL SFNC • Renamed of TLTYPE USB3 to U3V • Added functions to retrieve the parent modules • Added DEVICE_INFO_ commands • Added PORT_INFO_ command • Added Pixel Endianness • Added numeric constants for infinite timeouts and invalid handles. • Added clarification that SchemaVersion as part of the URL is only to be used with legacy GCGetPortURL function • Added BUFFER_INFO_DATA_SIZE and explanation • Added ZIP clarification • Added BUFFER_INFO_TIMESTAMP_NS and DEVICE_INFO_TIMESTAMP_FREQUENCY • Changed ‘revision’ to ‘version’ • Clarification of STREAM_INFO_NUM_DELIVERED • Added reference to SFNC Transfer Control features. • Clarification on Module enumeration issues • Added reference to GenTL SFNC • Added PFNC to PixelFormat Namespaces • Extended return code information for GenTL functions • Added UTF8 encoding

			<ul style="list-style-type: none"> • Added return code descriptions • Renamed EVENT FEATURE DEVEVENT to EVENT REMOTE DEVICE • Added EVENT MODULE • Added GC ERR INVALID VALUE • Deprecated PAYLOAD TYPE EXTENDED CHUNK and changed comment on PAYLOAD TYPE CHUNK DATA • Added GC ERR RESOURCE EXHAUSTED and GC ERR OUT OF MEMORY • Added error codes to function descriptions • Clarified number of images to acquire in DSStartAcquisition • Added clarification to EventKill function • Clarified handling of too small buffers • Clarified the retrieval of the payload size from the GenTL Producer • Clarified the behavior of EventKill/EventGetData
1.5	September 2014	GenTL Subcommittee	<p>Changes for V. 1.5</p> <ul style="list-style-type: none"> • Changed Standard Feature Naming Convention to Standard Features Naming Convention • Added URL INFO commands for register address and files size • Added methods and enumerations to handle multi-part buffers • Added URL INFO commands to retrieve scheme and filename • Added error code GC ERR BUSY • Added access status for already open devices • Introduced Mandatory and Optional for Info Commands
1.6	October 2019	GenTL Subcommittee	<p>Changes for V. 1.6</p> <ul style="list-style-type: none"> • Added functions and enumerations to handle data stream flows and composite buffers as their destinations; new API functions: DSAnnounceCompositeBuffer (plus related command BUFFER INFO IS COMPOSITE), DSGetNumFlows, DSGetFlowInfo, DSGetNumBufferSegments, DSGetBufferSegmentInfo • Added support for composite buffers in acquisition engine and existing buffer related content

			<ul style="list-style-type: none"> • Added support for GenDC payload type (PAYLOAD_TYPE_GENDC, STREAM_INFO_FLOW_TABLE, STREAM_INFO_GENDC_PREFETCH_DESCRIPTOR) • Extensions to GenTL multi-part definition to fully cover multi-part in GigE Vision 2.1 (BUFFER_PART_INFO_REGION_ID, BUFFER_PART_INFO_DATA_PURPOSE_ID, PART_DATATYPE_JPEG, PART_DATATYPE_JPEG2000) • Clarified relationships between BUFFER_INFO_SIZE_FILLED and BUFFER_INFO_DATA_SIZE • Clarified several error return conditions • Clarified interpretation of partial port reads/writes as failure • Further clarified use of BUFFER_INFO_NEW_DATA and STREAM_INFO_BUF_ALIGNMENT • Added support for „stacked“ buffer info querying, DSGetBufferInfoStacked and DSGetBufferPartInfoStacked
--	--	--	---

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 Features 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 GenTL Subcommittee

The GenTL Subcommittee is part of the GenICam Standard Group hosted by the EMVA.

1.3 Acronyms and Definitions

1.3.1 Acronyms

Term	Description
CL	Camera Link
CTI	Common Transport Interface
GenApi	GenICam Module
GenICam	Generic Interface to Cameras
GenTL	Generic Transport Layer
GenTL SFNC	GenICam Module: GenTL Standard Features Naming Convention
SFNC	GenICam Module: Standard Features Naming Convention
PFNC	GenICam Module: Pixel Format Naming Convention
GenDC	GenICam Module: Generic Data Container

GEN<i>i</i>CAM		 emva
Version 1.6	GenTL Standard	

Term	Description
GigE	Gigabit Ethernet
IIDC	1394 Trade Association Instrumentation and Industrial Control Working Group, Digital Camera Sub Working Group
PC	Personal Computer
SFNC	GenICam Module: Standard Features Naming Convention
TLI	Generic Transport Layer Interface
USB	Universal Serial Bus
UVC	USB Video Class

1.3.2 Definitions

Term	Description
Configuration	Configuration of a module through the GenTL Port functions, a GenApi compliant XML description and the GenTL Standard Features Naming Convention
GenApi	GenICam module defining the XML Schema which is used to describe register maps
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.
TLParamsLocked	XML-Feature in the XML of the remote device to prevent the change of certain features during an acquisition

1.4 References

EMVA GenICam Standard	http://www.genicam.org
AIA GigE Vision Standard	http://www.machinevisiononline.org/
AIA USB3 Vision Standard	http://www.machinevisiononline.org/
JIA CoaXPress Standard	http://jiaa.org/en/
AIA Camera Link HS Standard	http://www.machinevisiononline.org/
AIA Camera Link Standard	http://www.machinevisiononline.org/
ISO C Standard (ISO/IEC 9899:1990(E))	
RFC 3986	Uniform Resource Identifier
RFC 1951	DEFLATE Compressed Data Format Specification v1.3

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 images 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.

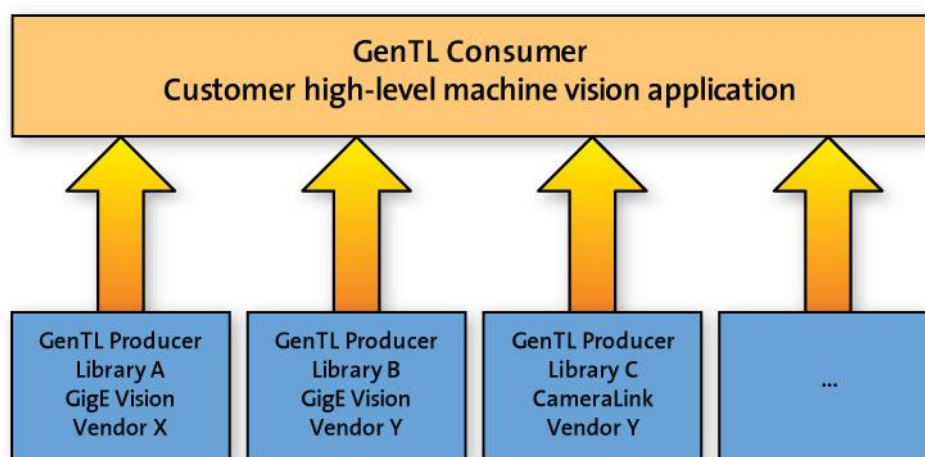


Figure 2-1: GenTL Consumer and GenTL Producer


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 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 that 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 a GenTL Producer implementation. If it is used internally, no access to it may be given through the C interface. Some reasons are:

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

- **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. The 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.

2.1.3 GenICam GenTL SFNC

In order to allow configuration of a GenTL Producer each module implements a virtual register map and provides a GenApi compliant XML file (see chapter [2.3.2](#)). Only mandatory features of these XML files are described in this document in chapter [7](#). All features (mandatory and non-mandatory) are defined in the GenTL SFNC document.

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.

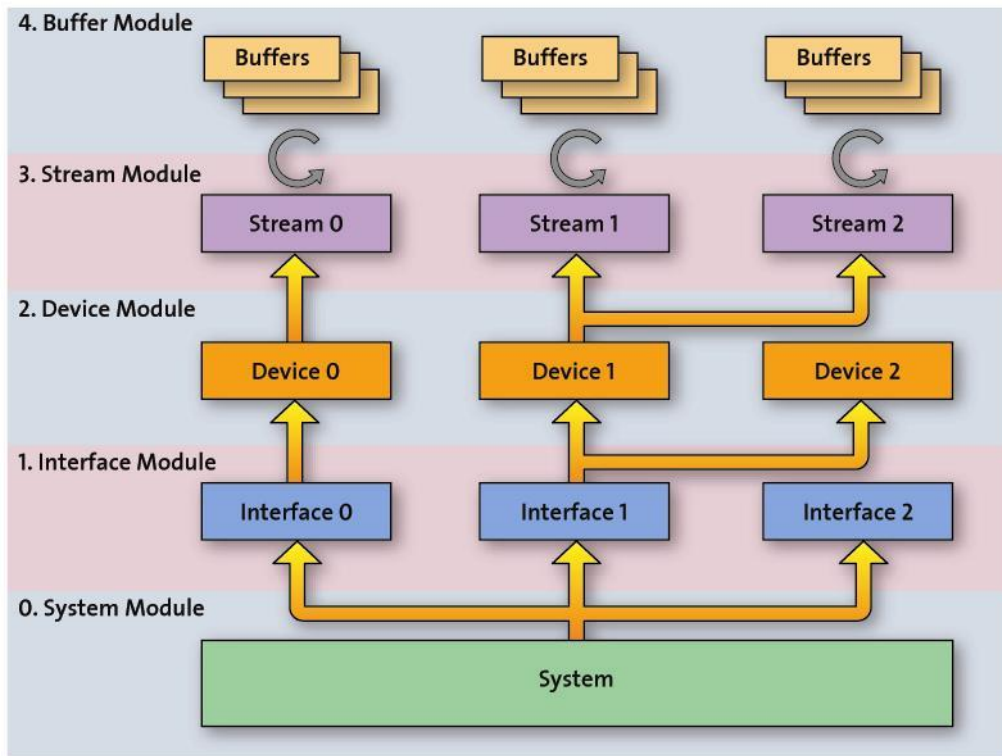


Figure 2-2: GenTL Module hierarchy

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 reported transport layer technology of the System module must be ‘Mixed’ (see chapter 6.6.1) 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; 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

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Interface module also presents Signaling and module configuration capabilities to the GenTL Consumer.

One system may contain zero, one or multiple interfaces. An interface represents only one transport layer technology. It is not allowed to have, e.g., a GigE Vision 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 GenTL Consumer 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.

The GenTL Producer may implement preprocessing of the image data which changes image format and/or buffer size. Please refer to chapter [5.5](#) for a detailed list of the parameters describing the buffer.

The buffer memory might be contiguous or segmented, as described in [5.7.1](#).

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.

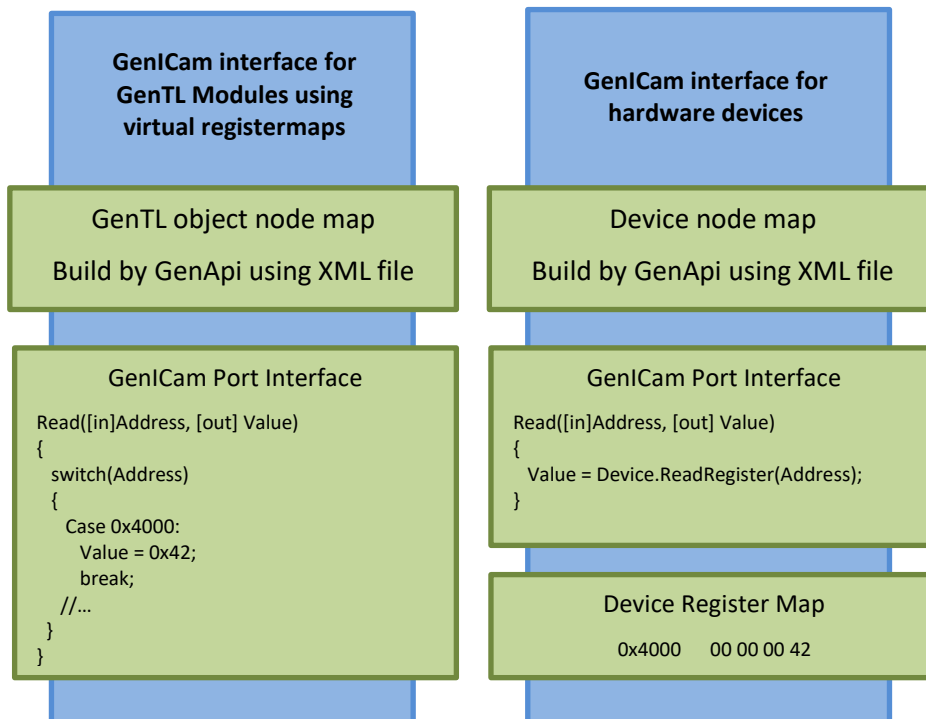


Figure 2-3: GenICam GenTL interface (C and GenApi Feature-interface)


The GenTL Producer driver consists of three logical parts: the C interface, the Configuration interface and the Event interface (signaling). The interfaces are detailed in the following paragraphs.

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. The default state of a GenTL Producer should ensure the ability to open a device and receive data from it.

A C interface was chosen because of the following 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).

GEN<i>CAM		
Version 1.6	GenTL Standard	

- **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 that 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 ANSI 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 a 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 that 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 New Buffer 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, but is encapsulated in the C interface.

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 59ff). For how to configure a certain module or get notified on events see chapter 4 Configuration and Signaling (page 31ff).

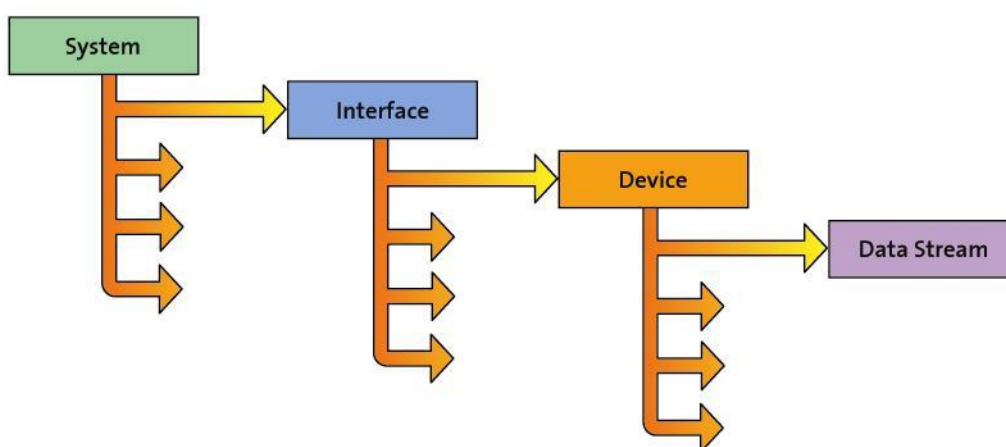



Figure 3-4: Enumeration hierarchy of a GenTL Producer

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 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.

There is no reference counting within a single process for [GCInitLib](#). Thus even when [GCInitLib](#) is called twice from within a single process space without accompanying call to [GCCloseLib](#), the second call will return the error [GC_ERR_RESOURCE_IN_USE](#). The first call to [GCCloseLib](#) from within that process will free all resources. The same is true for multiple calls to [GCCloseLib](#) without an accompanying call to [GCInitLib](#).

GEN<i>CAM		
Version 1.6	GenTL Standard	

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 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 is made to create a second System module instance from another operating system process.

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 the 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. Any call to [TLUpdateInterfaceList](#) does not affect instantiated interface handles. It may only change the order of the internal list accessed via [TLGetInterfaceID](#). The instantiation of a child interface with a known id is possible without a previous enumeration. It is recommended to call [TLUpdateInterfaceList](#) after reconfiguration of the System module to reflect possible changes.

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 done in a thread safe way.

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_HANDLES` 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 and time.

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

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

directly open an interface without inquiring the list of available interfaces via [TLUpdateInterfaceList](#). 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. [TLUpdateInterfaceList](#) must be called by the GenTL Consumer before any call to [TLGetNumInterfaces](#) or [TLGetInterfaceID](#). After successful module instantiation the [TLUpdateInterfaceList](#) function may be called by the GenTL Consumer so that it is aware of any change in this list. For convenience 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 requests the ID of a module, the GenTL Producer must return its 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.

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 which can differ from the order they have been instantiated in. The module does no reference counting. If an Interface module handle is requested a second time from within one process space the second call will return the error [GC_ERR_RESOURCE_IN_USE](#). A single 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 by a System module wide unique ID and not by the index. 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. It is recommended to call [IFUpdateDeviceList](#) regularly from time to time and after reconfiguration of the Interface module to reflect possible changes.

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

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 done in a thread safe way because the access to the lists could be made from multiple threads and the storage for these lists is not thread local. Therefore updating the list from one thread can affect the index used in another thread.

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_HANDLES 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 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 by calling [IFOpenDevice](#). 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. [IFUpdateDeviceList](#) must be called before any call to [IFGetNumDevices](#) or [IFGetDeviceID](#). In case the instantiation of a Device module is possible without having an internal device list the [IFOpenDevice](#) may be called without calling [IFUpdateDeviceList](#) before. This is necessary if the devices cannot be enumerated in a system, e.g., a network interface with a GigE Vision 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. A call to [IFUpdateDeviceList](#) does not affect any instantiated Device modules and its handles, only the order of the internal list may be affected.

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 GenTL Producer driver to instantiate the Device module instead of using the unique ID.

After successfully opening the Device using [IFOpenDevice](#), the device module is assumed to be fully operational, based on a real connection to concrete remote device (see also chapter 3.4). There might be situations when the host side of the connection (i.e. the Interface module) needs to be configured to successfully open the device (examples can be correct IP address setting of a network card matching the connected GigE Vision camera or frame grabber bit rate configuration). In such cases the GenTL Producer might provide suitable features to configure the connection parameters in the Interface module nodemap. GenTL Consumer can use them when necessary before calling [IFOpenDevice](#). The set of such

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

features is specific to each respective technology and in many cases no specific configuration is required at all.

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 an 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.

If a GenTL Consumer requests a Device that has been instantiated from within the same process beforehand and has not been closed, the Interface returns an error. If the instance was created in another process space and the GenTL Producer explicitly wants to grant access to the Device this access should be restricted to read only. The module does no reference counting within one process space. If a Device module handle is requested a second time from within one process space, the second call will return the 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. 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. As with the Interface and the Device lists, this list holds the unique IDs of the available streams. The number of data streams or the data stream IDs may not change during a single session. The IDs of the data streams must be fix between all sessions.

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.

GEN<i>CAM		
Version 1.6	GenTL Standard	

In case a given GenTL Producer does not provide a data stream it must return “0” for the number of available stream channels. In this case a call to [DevOpenDataStream](#) and all data stream related functions which start with a DS in the name will fail. This is then called a “Non Streaming Implementation”. It only covers the control layer which is responsible for enumeration and communication with the device.

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 [42ff](#)). 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.

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 a second time from within one process space the second call will return the 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](#), [DSAllocAndAnnounceBuffer](#) or [DSAnnounceCompositeBuffer](#) 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](#) or [DSAnnounceCompositeBuffer](#) 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 (consumer vs. producer allocated buffers) must not be mixed on a single Data

GEN<i><i></i>CAM		
Version 1.6	GenTL Standard	

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 the same buffer is announced twice on a single stream via a call to [DSAnnounceBuffer](#) the error `GC_ERR_RESOURCE_IN_USE` is returned. A buffer may be announced to multiple streams. In this case individual handles for each stream will be returned. In general there is no synchronization or locking mechanism between two streams defined. A GenTL Producer may though provide special functionality to prevent data loss. In case a GenTL Producer is not able to handle buffers announced to multiple streams it may refuse the announcement and return `GC_ERR_RESOURCE_IN_USE`.

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 buffer announcement functions.

A `BUFFER_HANDLE` retrieved either by [DSAnnounceBuffer](#), [DSAllocAndAnnounceBuffer](#) or [DSAnnounceCompositeBuffer](#) 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 to a single stream.

The more advanced method of announcing “composite” buffers using [DSAnnounceCompositeBuffer](#) is described in [5.7.1](#).

3.7 Enumerated modules list overview


The purpose of this chapter is to highlight possible issues related to the maintenance of the list of GenTL modules (interfaces, devices) available in a system. It provides a summary of principles listed in other chapters of the specification.

While the set of Data Stream modules implemented by a device is static and stays fixed throughout the lifetime of the local GenTL Device module, the lists of interfaces within a system and devices discovered on an interface are dynamic and might be updated on request by the GenTL Consumer.

The explicit request to update the list might be issued through the C interface ([TLUpdateInterfaceList](#) and [IFUpdateDeviceList](#) functions) or through corresponding commands (`InterfaceUpdateList`, `DeviceUpdateList`) of the parent module.

It’s important to remark that there might be multiple different views of the list of “currently available” modules, which we’ll demonstrate on an example of devices discovered on an interface:

- Real devices that are physically connected to the interface. If a new device is connected at runtime (or powered up), the GenTL Producer might or might not be aware of it. This depends on whether it actively monitors the interface. But it will not be published to the

GEN<i>CAM		
Version 1.6	GenTL Standard	

GenTL Consumer through the C interface nor the nodemap, until the Consumer explicitly requests to update the list. Similarly, if the device gets physically disconnected (or powered off), it will not be removed from the list published to the GenTL Consumer ([IFGetNumDevices/DeviceSelector](#)) until the next list update is executed.

- List of devices discovered on a given interface at the time of the last request to update the device list ([IFUpdateDeviceList](#) function or DeviceUpdateList command in the nodemap) and published to the GenTL Consumer through the C interface ([IFGetNumDevices](#)) and the nodemap (DeviceSelector). While the GenTL Producer maintains just a single list and publishes it identically through both interfaces the two views might still temporarily differ from the GenTL Consumer's viewpoint. If the list is updated from the nodemap (using DeviceUpdateList command), it is reflected by the nodemap directly through the C interface. If the list is updated from the C interface ([IFUpdateDeviceList](#) function), it might not be reflected by the nodemap directly due to GenApi caching effects. Finally, both views (C interface and nodemaps) might be used by the GenTL Consumer independently. It might be querying information through the C interface about one device, while the user selected (DeviceSelector) a different one in the nodemap.
- Currently opened local device modules, e.g., modules for which the GenTL Consumer owns valid handles ([IFOpenDevice](#)). This is typically a subset of the list published through the C interface and the nodemap. However, the specification requires that instantiated handles are not affected by any list update requests. This means that if a device is physically disconnected at runtime (while the consumer owns a valid handle for it), the handle remains valid, until explicitly closed ([DevClose](#)) – even if most operations upon that handle would simply fail. A request to update the device list would, remove such a device from the list published by the parent interface. A module handle becomes implicitly invalid whenever its parent (or grandparent) module is closed. Please note that the specification allows to open the device ([IFOpenDevice](#), similarly for interfaces) directly using a known device ID (the ID's should be unique and must not change between sessions) without calling [IFUpdateDeviceList](#) first. In this case the GenTL Producer might need to (re)execute the device discovery process on its own to connect to the device, providing the handle to the GenTL Consumer while the published device list remains unchanged (possibly even empty) until next list-update request.

3.8 Example

This sample code shows how to instantiate the first Data Stream of the first Device connected to the first Interface. Error checking is omitted for clarity reasons.

3.8.1 Basic Device Access

Functions used in this section are listed in subsequent sections.

```
{
    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( );
}

```

3.8.2 InitLib

Initialize GenTL Producer.

```

void InitLib( void )
{
    GCInitLib( );
}

```

3.8.3 OpenTL

Retrieve TL Handle.

```

TL_HANDLE OpenTL( void )
{
    TLOpen( hTL );
}

```

3.8.4 OpenFirstInterface

Retrieve first Interface Handle.

```

IF_HANDLE OpenFirstInterface( hTL )
{
    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 );
    }
}

```

```

    return hNewIface;
}
}

```

3.8.5 OpenFirstDevice

Retrieve first Device Handle.

```

DEV_HANDLE OpenFirstDevice( hIF )
{
    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 );
        return hNewDevice;
    }
}

```

3.8.6 OpenFirstDataStream

Retrieve first data Stream.

```

DS_HANDLE OpenFirstDataStream( hDev )
{
    // 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
        DevOpenDataStream( hDev, StreamID, hNewStream );
    }
}

```

3.8.7 CloseDataStream

Close Data Stream.

```

void CloseDataStream ( hStream )
{

```

```
DSClose( hStream );  
}
```

3.8.8 CloseDevice

Close Device.

```
void CloseDevice( hDevice )  
{  
    DevClose( hDevice );  
}
```

3.8.9 CloseInterface

Close Interface.

```
void CloseInterface( hIface )  
{  
    IFClose( hIface );  
}
```

3.8.10 CloseTL

Close System module.

```
void CloseTL( hTL )  
{  
    TLClose( hTL );  
}
```

3.8.11 CloseLib

Shutdown GenTL Producer.

```
void CloseLib( void )  
{  
    GCCloseLib( );  
}
```

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

4 Configuration and Signaling

Every module from the System to the Data Stream supports a GenTL Port for the configuration of the module's internal settings and the Signaling to the calling GenTL Consumer. For the Buffer module the GenTL Port is optional.

For a detailed description of the C function interface and data types see chapter 6 Software Interface (page 59ff). 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 and Instantiation (page 20ff).

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 or writes a block of data 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 Features Naming Convention for GenTL (GenTL SFNC) (page 180ff).

The Port functions of the C interface include a [GCReadPort](#) function and a [GCWritePort](#) function which can be used to implement an IPort object for the GenApi implementation. These functions resemble the IPort `Read` and `Write` functions in their behavior.

Register access through the Port functions is always byte aligned. In case the underlying technology does not allow byte aligned access the GenTL Producer must simulate that by reading more bytes than requested and returning only the requested bytes and by doing a read/modify/write access to the ports register map.

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.

GEN<i><i></i>CAM		
Version 1.6	GenTL Standard	

The URL containing the location of the according GenICam XML description can be retrieved through calls to the [GCGetNumPortURLs](#) and [GCGetPortURLInfo](#) functions 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 endianness or the allowed access (read/write, read only,...).

Two modules are special in the way the Port access is handled and are described in the following chapters.

4.1.1.1 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.

4.1.1.2 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 last 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 [GCGetNumPortURLs](#) function and the [GCGetPortURLInfo](#) 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 for GenTL Producer)
- Local Directory
- Vendor Web Site

A GenTL Consumer is required to implement 'Module Register Map' and 'Local Directory'. The download from a vendor's website is optional.

Supported formats are:

- Uncompressed XML description files
- Zip-compressed XML description files. The compression methods used are DEFLATE and STORE as described in RFC 1951.

4.1.2.1 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 SchemaVersion 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. This optional SchemaVersion is only to be used with the legacy function [GCGetPortURL](#). For current implementations the [GCGetPortURLInfo](#) function is used to obtain the SchemaVersion for a specific XML file. 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
<i>local</i>	Indicates that the XML description file is located in the virtual register map of the module.
<i>filename</i>	Information file name. It is recommended to put the vendor, model/device and version information in the file name separated by an underscore. For example: ‘tlguru_system_rev1’ for the first version of the System module file of the GenTL Producer company TLGuru.
<i>extension</i>	Indicates the file type. Allowed types are <ul style="list-style-type: none"> • ‘xml’ for an uncompressed XML description file. • ‘zip’ for a zip-compressed XML description file.
<i>address</i>	Start address of the file in the virtual register map. It must be expressed in hexadecimal form without a prefix.
<i>length</i>	Length of the file in bytes. It must be expressed in hexadecimal form without a prefix.
<i>SchemaVersion</i>	Version the referenced XML complies to. The version is specified in major.minor.subminor format. This only concerns the legacy GCGetPortURL function since the legacy mechanism has no other ways to report a SchemaVersion for the XML file. For the new GCGetPortURLInfo function the SchemaVersion should be retrieved through the info commands.

A complete local URL would look like this:

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

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

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.

4.1.2.2 Local Directory

URLs in the form “*file:///filepath.extension[?SchemaVersion=1.0.0]*” or “*file:filename.extension[?SchemaVersion=1.0.0]*” 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 could be used to determine the location using a file dialog for example.

In order to comply with some Windows notations it is also allowed to replace the ‘*|*’ after the drive letter with a ‘*:*’.

It is recommended to put the vendor, model or device and version information in the file name separated by an underscore. For example: *tlguru_system_rev1* for the first version 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

4.1.2.3 Vendor Web Site (optional)

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 version information in the file name separated by an underscore. For example: *tlguru_system_rev1* for the first version 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

This sampel shows how to retrieve the Port module xmls.

```
{
// Retrieve the number of available URLs
GCGetNumPortURLs( hModule, NumURLs );
for( i=0; i<NumURLs; i++ )
{
    URLSize = 0;
    GCGetPortURLInfo( hModule, i, URL_INFO_URL, 0, 0, &URLSize );

    // Retrieve an string buffer to store the URL
    GCGetPortURLInfo( hModule, i, URL_INFO_URL, 0, pURL, &URLSize );

    if ( ParseURLLocation( pURL ) == local )
    {
        // Retrieve the address within the module register map from the URL
        Addr = ParseURLLocalAddress( pURL );
        Length = ParseURLLocalLength( pURL );
        // 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 the acquisition engine for example 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 occurs, the event object is signaled or the wait is terminated by the GenTL Consumer. 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 (in the table below), it does not support that event type.

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, which retrieves the data associated with the event.

There are no mandatory event types. If an event type is not implemented in a GenTL Producer the [GCRegisterEvent](#) should return [GC_ERR_NOT_IMPLEMENTED](#). If an event type is implemented by a GenTL Producer module it is recommended to register an event object for that event type. The event types are described in the following table.

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. For example this event applies to an error while data is acquired in the acquisition engine of a Data Stream module.
New Buffer	Data Stream	New data is present in a buffer in the acquisition engine. In case the New Buffer event is implemented it must be registered on a Data Stream module. After registration the calling GenTL Consumer is informed about every new buffer in that stream. If the EventFlush function is called all buffers in the output buffer queue are discarded. If a DSFlushQueue is called all events from the event queue are removed as well. Please use the BUFFER_INFO_IS_QUEUED info command in order to inquire the queue state of a buffer.

Event Type	Modules	Description
Feature Invalidate	Local Device	<p>This event signals to a calling GenTL Consumer that 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.</p> <p>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. The use of this mechanism implies that the user must make sure that all terminal nodes the feature depends on are invalidated in order to update the GenApi cache. The provided feature name may not be standardized in SFNC. In case the feature is covered through SFNC the correct SFNC name should be used by the GenTL Producer. In case the provided feature name is under a selector the GenTL Consumer must walk through all selector values and invalidate the provided feature and all nodes it depends on for every selector value.</p>
Feature Change	Local Device	<p>This event communicates to a GenTL Consumer that a GenApi feature must be set to a certain value. This is for now only intended for the use in combination 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. Future use cases might be added when appropriate.</p> <p>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.</p>
Remote Device Event	Local Device	<p>This event communicates to a calling GenTL Consumer that a GenApi understandable event occurred, initiated by the Remote Device. The event ID and optional data delivered with this event can be put into a GenApi Adapter which then invalidates all related nodes.</p> <p>This event used to be called Feature Device Event but has been renamed in order to be in sync with the enumeration for the event type.</p>
Module Event	All	<p>This event communicates to a calling GenTL Consumer that a GenApi understandable event</p>

GEN<i>CAM		
Version 1.6	GenTL Standard	

Event Type	Modules	Description
		occurred, initiated by the GenTL Producer module the event was registered on. The event ID and the 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 dequeue operations from multiple threads.

An event object's state 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 the event object's state handling must be atomic.

4.2.3 Event Handling

The handling of the event objects is always the same independently of the event type. The signal reason and the signal data of course depend on the event type. The complete state handling is done by the GenTL Producer driver. The GenTL Consumer may call the [EventKill](#) function to terminate a single instance of a waiting [EventGetData](#) operation. This means that if more than one thread waits for an event, the [EventKill](#) function terminates only one wait operation and other threads will continue execution.

4.2.3.1 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 the error [GC_ERR_RESOURCE_IN_USE](#).

To unregister an event object the [GCUnregisterEvent](#) function must be called. If a module is closed all event registrations are automatically unregistered. Events that are still in the queue while an event object is unregistered are silently discarded. Pending wait operations

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

through calls to [EventGetData](#) are terminated with a [GC_ERR_ABORT](#) when the event object is unregistered through [GCUnregisterEvent](#).

After an `EVENT_HANDLE` is obtained the GenTL Consumer can wait for the event object to be signaled by calling the [EventGetData](#) function. 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.

4.2.3.2 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 or, if no data is available waits for an event being signaled with the provided timeout. 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, [EventKill](#) is issued or until the next event data becomes available. If [EventKill](#) is called exactly one call to [EventGetData](#) will return [GC_ERR_ABORT](#) even if [EventKill](#) is called while no [EventGetData](#) call was waiting. Also the return of [GC_ERR_ABORT](#) has higher priority than delivering the next event from the queue so that even if there are one or more events in the queue ready to be delivered to the user through a call to [EventGetData](#), after a call to [EventKill](#) the next call to [EventGetData](#) will return [GC_ERR_ABORT](#). In this case no event is removed from the queue and no data is delivered to the GenTL Consumer. The counter [EVENT_NUM FIRED](#) is not affected by the calls to [EventKill](#).

In case an event object is unregistered through a call to [GCUnregisterEvent](#), it's previous state will be lost. This also applies to previous calls to the [EventKill](#) function. When re-registering an event through a call to [GCRegisterEvent](#) on this port later on [EventGetData](#) will not return [GC_ERR_ABORT](#) until [EventKill](#) is called again.

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 the next call to [EventGetData](#) will deliver the next queue entry. Otherwise the event object is reset to not signaled state. The maximum size of the buffer delivered through [EventGetData](#) can be queried using `EVENT_SIZE_MAX` with the [EventGetInfo](#) function. The GenTL Consumer must not perform data size queries since a call of [EventGetData](#) with a NULL pointer for the buffer will remove the data from the queue without delivering it. In this case the event is counted as if it would have been fired and the data is discarded.

The exact type of data is dependent on the event type and the GenTL Producer implementation. The data is copied into an user buffer allocated by the GenTL Consumer. The content of the event data can be queried with the [EventGetDataInfo](#) function. The maximum size of the buffer to be filled is defined by the event type and can be queried using [EVENT_INFO_DATA_SIZE_MAX](#) after the buffer is delivered. This information can be queried using the [EventGetInfo](#) function.

The events are handled as described in the following steps:

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

- Register a DeviceEvent on the corresponding GenTL module.
- Inquire the max needed buffer size.
- Allocate the buffer to receive the event data.
- Wait for the event and data. The structure of the data in the provided buffer is not defined and GenTL Producer dependent. The only exception to that would be the New Buffer event which provides a defined internal struct.
- Extract the data in the buffer using [EventGetDataInfo](#). This step is not necessary in cases when the GenTL Consumer knows the contents of the buffer delivered through [EventGetData](#), such as a New Buffer event.
- Data processing/usage.
- Unregister event.
- Deallocate buffer.

As described the content of the buffer retrieved through [EventGetData](#) is GenTL Producer implementation specific and may be parsed using the [EventGetDataInfo](#) function. The only exception to that is the New Buffer event which will return the [EVENT_NEW_BUFFER_DATA](#) structure.

For the Device Event events ([EVENT_REMOTE_DEVICE](#)) the GenTL Producer must provide two types of information about every single event, so that it can be "connected" to the remote device's nodemap:

- Event ID: queried through [EventGetDataInfo\(EVENT_DATA_ID\)](#). The ID is passed as a string representation of hexadecimal form, for example "CF51" without the leading '0x'. The ID can be also queried directly in numeric form using [EventGetDataInfo\(EVENT_DATA_NUMID\)](#).
- Event data: buffer containing the (optional) data accompanying the event. It must correspond with the data addressable from the remote device nodemap, the beginning of the buffer must correspond with address 0 of the nodemap's event port. For example, for GigE Vision devices this is by convention the entire EVENTDATA packet, without the 8-byte GVCP header.

Also for the module's events ([EVENT_MODULE](#)) the GenTL Producer must provide two types of information about every single event, so that it can be connected to a module's nodemap:

- Event ID: queried through [EventGetDataInfo\(EVENT_DATA_ID\)](#). The ID is passed as a string representation of hexadecimal form, for example "CF51" without the leading '0x'. The ID can be also queried directly in numeric form using [EventGetDataInfo\(EVENT_DATA_NUMID\)](#).
- Event data: buffer containing the (optional) data accompanying the event. It must correspond with the data addressable from the module's nodemap, the beginning of the buffer must correspond with address 0 of the nodemap's event port, similar to way the [EVENT_REMOTE_DEVICE](#) is working.

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Note: to improve interoperability, it is recommended that for device events based on "standard" event data formats, the buffer delivered through [EventGetData](#) is directly the buffer that can be fed to the corresponding standard GenApi event adapter. For example, in case of GigE Vision it would be the entire EVENTDATA packet including the header.

If queued event data is not needed anymore the queue can be emptied by calling the [EventFlush](#) function on the associated EVENT_HANDLE. To inquire the queue state of a buffer the GenTL Consumer can call [DSGetBufferInfo](#) with the info command [BUFFER_INFO_IS_QUEUED](#).

Signals that occur without a corresponding event object being registered using [GCRegisterEvent](#) are silently discarded.

A single event notification carries one event and its data.

For example, a GigE Vision device event sent through the message channel carrying multiple EventIDs in a single packet must result in multiple GenTL Producer events. Each GenTL Producer event will then provide a single GigE Vision EventID.

4.2.4 Example

This sample shows how to register a New Buffer event.

```
{
  GCRegisterEvent(hDS, EVENT_NEW_BUFFER, hNewBufferEvent);
  CreateThread ( AcqFunction );
}
```

4.2.4.1 AcqFunction

This sample shows the wait loop to retrieve new buffers.

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

GEN<i><i></i>CAM		 emva
Version 1.6	GenTL Standard	

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 data transfer mechanism so that it can be used, if not for all, then for most 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). It depends on the GenTL Producer if a buffer may be announced during an ongoing acquisition (see [5.2.2](#)). A buffer will stay in this pool even when it is referenced from other queues/pools like the Input Buffer Pool (see [5.1.2](#)) or the Output Buffer Queue (see [5.1.3](#)) or when it is delivered to the GenTL Consumer until it is revoked.

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 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 successfully filled, it is put into the output buffer queue. If the transfer was not successful or if the acquisition has been stopped with [ACQ_STOP_FLAGS_KILL](#) specified the buffer is placed into the output buffer queue by default. It is up to the implementor to provide additional buffer handling modes which would handle that partially filled buffer differently.

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

GEN<i>CAM		
Version 1.6	GenTL Standard	

buffer is removed from the output buffer queue (delivered), the acquisition engine must not write data into it. Thus this is 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 buffer handling 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 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

Note: this section describes the acquisition chain based on the “traditional” approach using linear contiguous buffers, available since first versions of GenTL specification. This approach is backwards compatible and works well for all payload types. To learn about the more advanced option using structured “composite buffers”, refer to chapter 5.7.

The following description shows the steps to acquire an image from the GenTL Consumer’s point of view (default buffer handling mode). Image or data acquisition is performed on the Data Stream module with the functions using the DS_HANDLE. Thus before an acquisition can be carried out, an enumeration of a Data Stream module has to be performed (see chapter 3 Module Enumeration (page 20ff). For a detailed description of the C functions and data types see chapter 6 Software Interface (page 59ff).

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.

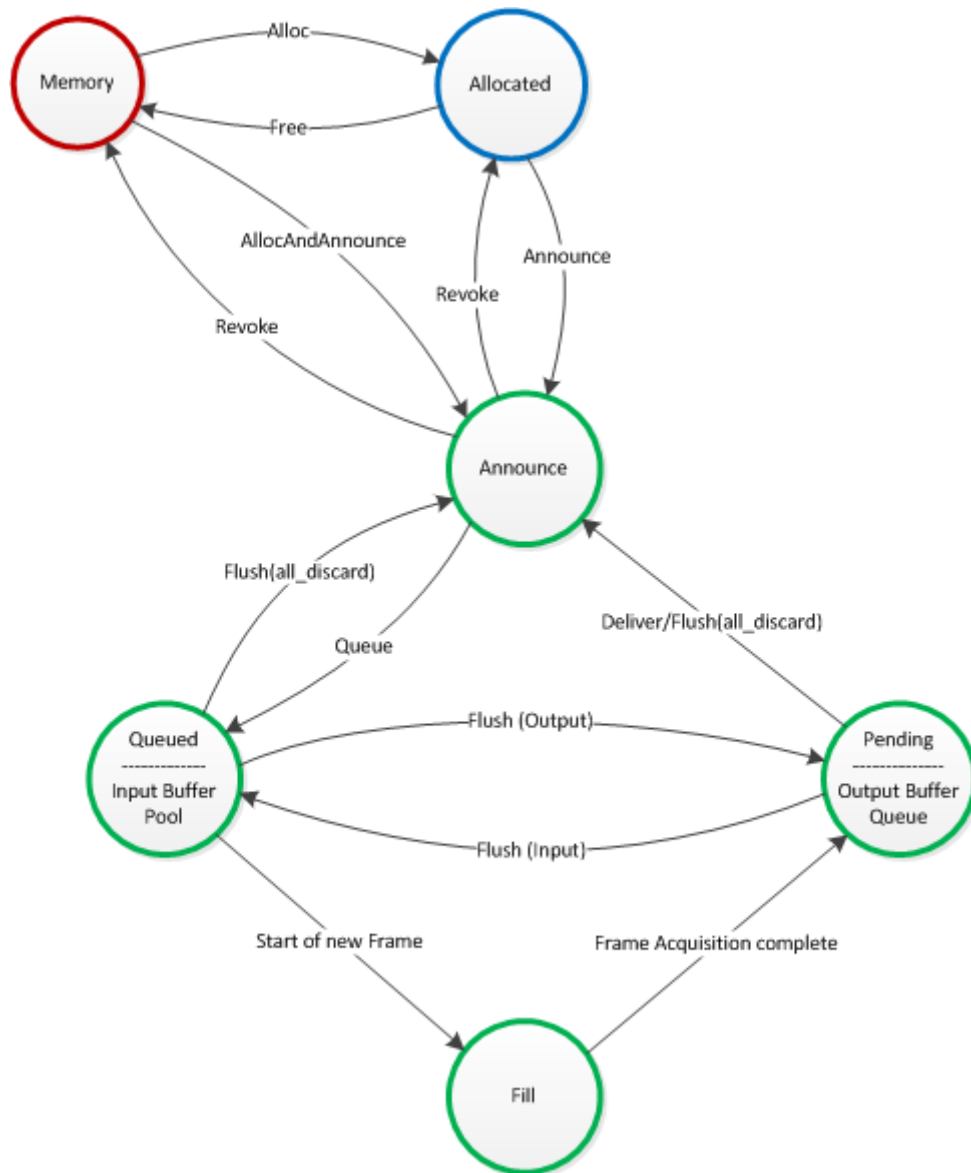


Figure 5-5: Acquisition chain seen from a buffer’s perspective

5.2.1 Allocate Memory

First the size of a single buffer has to be obtained. In order to obtain that information the GenTL Consumer must query the GenTL Data Stream module (important: not the remote device) to check if the payload size information is provided through the GenTL Producer by calling [DSGetInfo](#) function with the command [STREAM INFO DEFINES PAYLOADSIZE](#). If the returned information is true the Consumer must call [DSGetInfo](#) with [STREAM INFO PAYLOAD SIZE](#) to retrieve the current payload size. Additionally the GenTL Producer may provide a “PayloadSize” feature in the node map of the Data Stream Module reflecting the GenTL Producer’s payload size. The value reported through that feature must be the same as provided through [DSGetInfo](#).

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

In case the returned information of [DSGetInfo](#) with [STREAM INFO DEFINES PAYLOADSIZE](#) is false the Consumer needs to inquire the PayloadSize through the node map of the remote device. The remote device port can be retrieved via the [DevGetPort](#) function from the according Device module. The GenTL Consumer has to select the streaming channel in the remote device and read the “PayloadSize” standard feature.

In any case the GenTL Producer together with the underlying technology must provide a way to retrieve the payload size. When the device does not provide the PayloadSize feature (for example in case of a GenTL Producer which is implementing an interface standard which is not specifying PayloadSize as a mandatory feature), the GenTL Producer itself must report the required payload size using stream info commands [STREAM INFO DEFINES PAYLOADSIZE](#) and [STREAM INFO PAYLOAD SIZE](#). Failure to query the required payload size would typically disallow the GenTL Consumer to set up the acquisition properly. It might try to calculate the payload size based on the device configuration, but such calculation would never be reliable.

If [STREAM INFO DEFINES PAYLOADSIZE](#) returns true the Data Stream module must provide the buffer describing parameters. This allows the GenTL Producer to modify the buffer parameters to preprocess an image. In case the GenTL Producer is doing that it must implement all buffer describing parameters. For a detailed description please refer to chapter 5.5.

With that information one or multiple buffers can be allocated as the GenTL Consumer needs. 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 module (if the error event is implemented on the Buffer module) and on the transmitting Data Stream module with a [GC ERR BUFFER TOO SMALL](#) error code. It is up to the GenTL Producer if a “too small” buffer is filled with parts of the transferred payload or if the buffer is not filled at all. In both cases the buffer should be delivered to the GenTL Consumer if the underlying technology allows it and the according [BUFFER INFO CMDs](#) [BUFFER INFO IS INCOMPLETE](#), [BUFFER INFO SIZE FILLED](#) and [BUFFER INFO DATA LARGER THAN BUFFER](#) should report the fill state. Also in case one or more of the announced buffers are smaller than the payload size the GenTL Producer can refuse to start the acquisition through [DSStartAcquisition](#) returning an error code [GC ERR BUFFER TOO SMALL](#).

The payload size for each buffer, no matter if defined by the GenTL Producer or by the remote device, may change during acquisition as long as the acquired payload size delivered is smaller than the actual payloadsize reported at acquisition start. The payload size of a given buffer can be queried through the [BUFFER INFO CMDs](#).

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. While usually all

buffers are announced prior to the call to [DSStartAcquisition](#) it is also possible to announce or revoke buffers in between calls to [DSStartAcquisition](#) and [DSStopAcquisition](#) while the acquisition is ongoing in case the underlying GenTL Producer supports this. In order to revoke a buffer it is additionally necessary that the particular buffer is only referenced in the announced buffer pool which means that it is neither in any of the acquisition queues and that it is currently not acquired to. In case the underlying GenTL Producer does not support the announcing or revoking buffers while the acquisition is active (in between calls to [DSStartAcquisition](#) and [DSStopAcquisition](#)) it is also valid for the GenTL Producer to return a `GC_ERR_BUSY` from a call to [DSAnnounceBuffer](#), [DSAllocAndAnnounceBuffer](#) or [DSRevokeBuffer](#).

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. The queue order also does not necessarily have an influence in which order the buffers are filled. This depends only on the buffer handling 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 the acquisition on the remote device is to be started by setting the “AcquisitionStart” standard feature via the GenICam GenApi.

If a device implements the SFNC Transfer Control features, the GenTL Consumer may need to start the transfer on the remote device as well, depending on the operating mode.

5.2.6 Acquire Image Data

The following action is performed in a loop:

- Wait for the “New Buffer” event to be signaled (see [4.2 Signaling](#) page [35ff](#))
- 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 if the buffer handling mode defines it.

Requeuing the buffers can be done in any order using 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. This mechanism locks the buffer effectively.

5.2.7 Stop Acquisition

When finished acquiring image data the acquisition on the remote device is to be stopped if necessary. This can be done by setting the “AcquisitionStop” 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. The same is true 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. If [ACQ_STOP_FLAGS_KILL](#) is specified in the call to [DSStopAcquisition](#) a partially filled buffer is by default moved to the output buffer queue for processing. [DSGetBufferInfo](#) with [BUFFER_INFO_IS_INCOMPLETE](#) would indicate that the buffer is not complete.

If a device implements the SFNC Transfer Control features, the GenTL Consumer may need to stop the transfer on the remote device, depending on the operating mode.

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 discards the buffers from the Output Buffer Queue.

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 cannot 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.

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

The order in which buffers can be revoked depends on the method in which they were 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. Nevertheless, it is recommended to keep the base pointer of a buffer for the lifetime of the buffer handle.

5.2.10 Free Memory

If the GenTL Consumer provides the memory for the buffers using the [DSAnnounceBuffer](#) function it also has to free it. Memory allocated by the GenTL Producer implementation using the [DSAllocAndAnnounceBuffer](#) function is freed by the library if necessary. The GenTL Consumer must not free this memory.

5.3 Buffer Handling Modes

Buffer handling modes describe the internal buffer handling during acquisition. There is only one mandatory mode defined in this document which GenTL Producer implementations should default to. More modes are defined in the GenICam GenTL Standard Features 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 in Figure 5-6 assumes that we are looking at 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 in the lower bar labeled 'User'. 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 cannot 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.

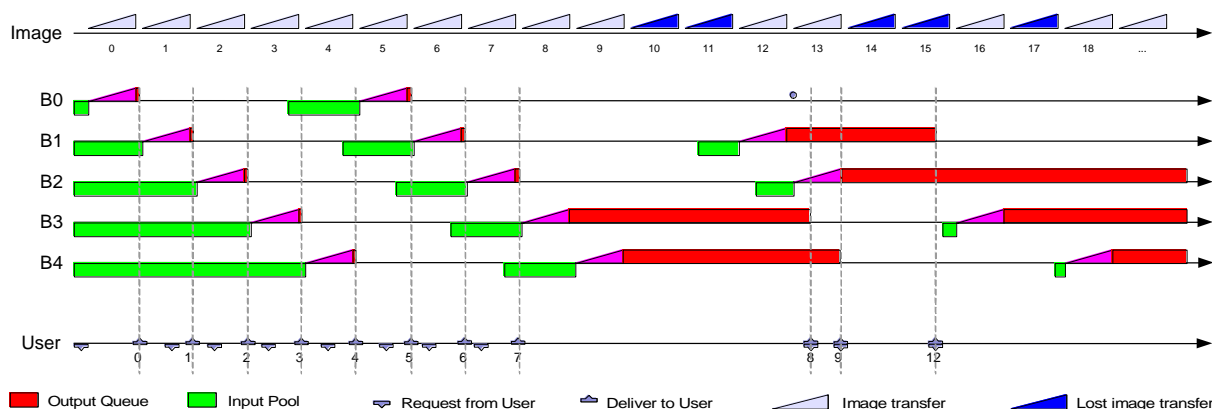


Figure 5-6: Default acquisition from the GenTL Consumer's perspective

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 requested.

Wrap-Up:

- There is 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.

5.4 Chunk Data Handling

5.4.1 Overview

The GenICam GenApi standard contains a notion of "chunk data". These are chunks of data present in a single buffer acquired from the device together with or without other payload type data. Each chunk is identified unequivocally by its ChunkID (up to 64Bit unsigned integer) which maps it to the corresponding port node in the remote device's XML description file. The information carried by individual chunks is described in the XML file. To address the data in the chunk the GenApi implementation must know the position (offset) of the chunk in the buffer and its size. The structure of the chunk data in the buffer is technology specific and it is therefore the responsibility of the GenTL Producer to parse the chunk data in the buffer (if there are any). To parse a buffer containing chunk data, the GenTL Consumer uses the function [DSGetBufferChunkData](#) which reports the number of chunks in the buffer and for each chunk its ChunkID, offset and size as an array of

[SINGLE_CHUNK_DATA](#) structures. This information is sufficient to connect the chunk to the remote device's nodemap (for example through the generic chunk adapter of GenApi reference implementation).

The acquired buffer might contain only the chunk data or the data might be mixed within the same buffer with an image or other data. To query, if a given buffer contains chunk data, the [BUFFER_INFO_CONTAINS_CHUNKDATA](#) command may be used which will return true in case the buffer contains chunk data or the function [DSGetBufferChunkData](#) can be queried which, in case the buffer contains accessible chunk data, would return the number of chunks available.

There are other chunk data related buffer info commands, such as [BUFFER_INFO_IMAGEPRESENT](#) (indicating that the buffer contains also an image) or [BUFFER_INFO_CHUNKLAYOUTID](#) (can help to check, if the chunk structure has changed since the last delivered buffer and if it is necessary to parse it again). The [STREAM_INFO_NUM_CHUNKS_MAX](#) command reports the maximum number of chunks to be expected in a buffer acquired through a given stream (if that maximum is known a priori).

If the GenTL Consumer knows the chunk data structure, such as accessing a device of a known standard technology, it is not necessary to use the [DSGetBufferChunkData](#) function to parse the buffer. The GenTL Consumer can use a more direct approach to extract the data (by using a standard chunk adapter in GenApi reference implementation).

5.4.1.1 Chunk data in composite buffer

The [DSGetBufferChunkData](#) function was introduced in GenTL with traditional contiguous buffers in mind, before introduction of composite buffers (announced using [DSAnnounceCompositeBuffer](#), 5.7.1). Because it treats the buffer as a single entity, it does not report the chunk offsets corresponding to a particular buffer segment, instead it reports linear offsets within the entire payload. The GenTL Consumer has to take this into account. If the format of the chunk data within the composite buffer is well known to the GenTL Consumer, it must parse the chunk data directly without help of [DSGetBufferChunkData](#), the GenTL Producer must return an error from that function.

5.4.1.2 Chunk data in a generic container (for example GenDC)

When the acquired payload contains data in a standard self-described container format (such as for example GenDC), the consumer must parse and interpret the chunk data directly according to the given container type specification, without help of the GenTL interface. GenTL Producer must return error ([GC_ERR_NO_DATA](#)) for all chunk data related queries, including [DSGetBufferChunkData](#).

5.4.2 Example

This sample shows how to retrieve chunk data from a buffer.

```
{
    // Check if the buffer contains chunk data
    DSGetBufferInfo (hStream, hBuffer, BUFFER_INFO_PAYLOADTYPE, Type, PayloadType,
        SizeOfPayloadType);
```

```

if ( PayloadType == PAYLOAD_TYPE_CHUNK_DATA )
{
    ChunkListSize = 0;
    DSGetBufferChunkData( hStream, hBuffer, 0, ChunkListSize )
    {
        // Alternatively it would be possible to inquire the max number of
        // chunks per buffer through STREAM_INFO_NUM_CHUNKS_MAX

        DSGetInfo( hStream, STREAM_INFO_NUM_CHUNKS_MAX, Type, ChunkListSize,
            sizeof(ChunkListSize));

        // In this case the consumer needs error checking in case the
        // GenTL Producer cannot provide that information
    }

    // Allocate array of SINGLE_CHUNK_DATA structures
    DSGetBufferChunkData( hStream, hBuffer, ChunkArray, ChunkListSize )

    // Pass Chunk Array to GenApi Port
    // Free ChunkArray.
}
}

```

5.5 Data Payload Delivery

The GenTL Producer is allowed to modify the image data acquired from the remote device if needed or if it is convenient for the user. An examples of such modifications can be a PixelFormat conversion (e.g., when decoding a Bayer encoded color image) or LinePitch adjustment (elimination of the line padding produced on the remote device).

Whenever a modification leads to a change of basic parameters required to interpret the image, the GenTL Producer must inform the GenTL Consumer about the modifications. It is mandatory to report the modified values through the [BUFFER_INFO_CMD](#) or [BUFFER_PART_INFO_CMD](#) commands of the C interface. The tables listing the values for [BUFFER_INFO_CMD](#) and [BUFFER_PART_INFO_CMD](#) also list which commands are optional and and which are mandatory.

If a given [BUFFER_INFO_CMD](#) command is not available, the GenTL Consumer assumes, that the GenTL Producer did not modify the corresponding parameter and that it corresponds to the settings on the remote device. For example, if the query for [BUFFER_INFO_PIXELFORMAT](#) returns an error, meaning that the [BUFFER_INFO_PIXELFORMAT](#) command is not available, the GenTL Consumer should assume that the GenTL Producer did

not modify the pixel format and that the pixel format in the buffer corresponds to the PixelFormat feature value in the nodemap of the remote device.

The only exception among the parameters listed in [BUFFER_INFO_CMD](#) and [BUFFER_PART_INFO_CMD](#) is the payload size value which needs to be known before any buffers are delivered (as it is used for buffer allocation). Thus, if the GenTL Producer modifies the payload size it has to report the actual value through the [STREAM_INFO_PAYLOAD_SIZE](#) command, as described in chapter 5.2.1.

It might be useful to report the modifications also through corresponding features of the stream and buffer nodemaps.

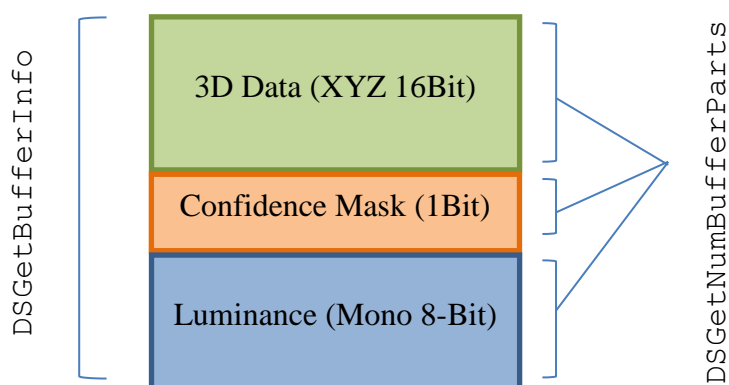
The GenTL Producer must take special care when modifying image data within a stream carrying chunk data. Such modifications must not result in a corrupted chunk data layout. In this case the GenTL Producer must reconstruct the chunk buffer.

5.6 Multi-Part Buffer Handling

5.6.1 Overview

There are many versatile use cases where the GenTL Producer needs to deliver different sets of data that belong logically together (in particular data coming from a single "exposure"), but consist of multiple distinct parts.

To allow effective delivery of such data the GenTL introduces a multi-part buffer payload type ([PAYLOAD_TYPE_MULTI_PART](#)). The different data segments of a multi-part buffer are placed physically into a single buffer. The number of the parts and the properties of the individual parts can be obtained using the functions [DSGetNumBufferParts](#) and [DSGetBufferPartInfo](#).



When receiving multi-part payload data, the GenTL Consumer is expected to query the number of distinct data parts in the buffer and properties of the individual parts using the functions mentioned above. It is important to note that some properties of the data (e.g., the AOI and/or the data format) are described as part-specific information using corresponding info commands supported by the [DSGetBufferPartInfo](#) function. When dealing with multi-part data the "buffer-global" info function [DSGetBufferInfo](#) can not be used to query buffer-part specific information (e.g., [BUFFER_INFO_PIXELFORMAT](#),

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

[BUFFER_INFO_WIDTH](#), [BUFFER_INFO_HEIGHT](#), etc.). Even if some of the properties, such as AOI size, are the same for all parts in a buffer, it should be reported and queried per-part via [DSGetBufferPartInfo](#), using the [BUFFER_PART_INFO_CMDS](#).

On the other hand buffer properties which are describing the global buffer and are not defined as part-specific have to be queried using [DSGetBufferInfo](#) (e.g., [BUFFER_INFO_TIMESTAMP](#), [BUFFER_INFO_NEW_DATA](#), [BUFFER_INFO_DELIVERED_CHUNKPAYLOADSIZE](#), etc). It is listed with the [BUFFER_INFO_XXX](#) constants which constant is overwritten by part specific information or if it describes the whole buffer.

Similar to any other basic payload type, it is possible to attach chunk data to the multi-part payload. The principles of chunk data handling remain the same as with other basic payload types. The chunk data is therefore common to all parts in the buffer. There is only one chunk section within a multi-part buffer.

The GenTL specification does not define strict rules for relationships between the individual data types within a buffer. Some typical use cases are discussed in the text below.

5.6.2 Planar Pixel Formats

A multi-part buffer can be used to reliably transfer and describe data using a planar pixel format such as the color data in separate R-G-B planes. In this case each part carries a single color plane. Typically all the parts share the same dimensions and differ only in the data format. For example, in this case the used data type would be (depending on the actual data) [PART_DATATYPE_2D_PLANE_TRIPLANAR](#).

With multi-part buffer all the planes that belong together are well and unambiguously described.

5.6.3 Multiple AOI's

There are devices which support multiple areas of interest (AOI's) to be captured within the sensor. The data from these multiple AOI's can also be effectively transferred using the multi-part payload. In this case the data format of all the parts is typically the same, the parts differ only in the AOI parameters.

It is up to the GenTL Consumer if it will prefer to treat the individual AOI's as independent images or if it will reconstruct a single image from the AOI's. The main advantage is that all the AOI's belonging to the same exposure are transferred together.

5.6.4 Pixel Confidence Data

A buffer part carrying the data type [PART_DATATYPE_CONFIDENCE_MAP](#) is used to identify levels of validity of the pixel values carried in other part(s). Each value in the confidence map specifies the confidence level of pixels at the same position (row/column) in other data part(s).

In the simplest case of a 1-bit confidence data the confidence map simply marks corresponding pixels valid or invalid. Higher bit depth integer data types in the validity map allow to specify the level of confidence ranging from 0 to the maximal value of given integer

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

data type. When using the floating point confidence format, the confidence level is usually reported using the interval [0.0, 1.0]. These rules are not necessarily strict and might be redefined in specific use cases.

Use cases for the pixel confidence data include assigning a level of reliability of individual point coordinates in a 3D point cloud or masking of non-rectangular images.

5.6.5 3D Data Exchange

3D devices frequently do not only provide the 3D data itself, but also additional information such as the intensity image, pixel confidence information or even various other additional image properties.

The multi-part payload type is thus usually the best option to transfer data belonging to a single exposure.

Note that to fully interpret the 3D data in a multi-part buffer, it is typically required to query additional information using the well-defined 3D data model in SFNC.

5.6.6 Non-Line Oriented Data


With some data formats (for example in case of a 3D point cloud), the pixels within the payload might not be necessarily line oriented or organized in a rectangular matrix, but rather just an unorganized set of pixels. In such case, it is recommended that the image width ([BUFFER PART INFO WIDTH](#)) is always set to 1 and image height ([BUFFER PART INFO HEIGHT](#)) is used to describe the number of unorganized pixels in the payload. This is aligned with similar practice in other standards.

5.6.7 Multi-Source Devices

In most "simple" cases the data in all parts originates from the same source (such as physical sensor) and can be pixel-mapped together. This means that pixels of the same row/column coordinates (considering also the AOI offset parameters of each part) are assumed to be expressing different properties of the same pixel in the acquired scene. This way, for example a point with given 3D coordinates coming from individual coordinate planes ([PART DATATYPE 3D PLANE TRIPLANAR](#)) can be mapped to the intensity value coming from pixel at the same position in the 2D image data part ([PART DATATYPE 2D IMAGE](#)).

There are, however, more complex devices providing possibly data from multiple sources in parallel. An example can be a dual-sensor device. In such case the pixels from parts carrying data from the different sources cannot be directly mapped together.

The producer reports the information which parts come from the same source (and thus can be pixel-mapped together) using the [BUFFER PART INFO SOURCE ID](#) info command. Data coming from same (pixel-mappable) source should be marked using the same source ID, data from different sources should be marked using different source ID's.

GEN<i>CAM		
Version 1.6	GenTL Standard	

5.7 Structured Data Acquisition

The universal acquisition into contiguous buffers (announced using [DSAllocAndAnnounceBuffer](#) or [DSAnnounceBuffer](#)) described in previous sections is sufficient in many use cases and ensures backward compatibility with older implementations.

However, GenTL specification version 1.6 introduces additional options allowing more flexible control of the acquisition engine: composite buffers and data stream flows.

5.7.1 Composite (Non-contiguous) Buffers

In some use cases, the GenTL Consumer might prefer that specific parts of the data acquired in each acquisition step (each “New Buffer” event or “block” as referred in some transport layer technologies) are stored in discrete memory locations.

To achieve that, the consumer announces each buffer as a set of memory segments, using [DSAnnounceCompositeBuffer](#) which returns a unique `BUFFER_HANDLE` similar to the other buffer announcement functions. From buffer flow and lifetime perspective the composite buffers are treated by the acquisition engine exactly same as the “traditional” contiguous buffers, they are queued, info-queried, signaled (through “New Buffer” events) or revoked same as other buffers (as described in [5.2](#)), the only difference is that they have an internal structure, consisting of discrete memory areas. To use them effectively, the GenTL Consumer and the GenTL Producer’s acquisition engine need to act and interpret the composite buffers’ structure in concert.

It is important to realize that announcing a composite buffer by itself is not sufficient to get the data properly split in its buffer segments during acquisition. The GenTL Consumer is responsible to care that the GenTL Producer and/or device are capable and configured for such operation.

The configuration of how the acquisition data are mapped into buffer segments is in general beyond scope of the GenTL specification and might be controlled for example by means of stream and data format related features in SFNC or GenTL SFNC. The composite buffers are primarily intended as targets for structured data acquired by data stream flows mechanism ([5.7.2](#)) and hence for acquisition in the GenDC format ([5.7.3](#)). In these cases, the mapping is implied by means of the flows and GenDC configuration.

5.7.2 Data Stream Flows

The acquisition data stream transported into the target buffers by means of a Data Stream module might be split into multiple “flows”. The flows are independent channels within the data stream, responsible to transfer individual components of the acquisition data.

The flow mechanism serves two main purposes:

- Allow transferring independent data parts into a single buffer in parallel as they become available, without need to buffer them within the source (remote device), which might be required for sequential transfer.

GEN<i><i></i>CAM		
Version 1.6	GenTL Standard	

- Allow transferring independent data parts into discrete target memory locations defined by the user or GenTL Consumer to optimize the possibly use case specific data handling (for example to route some data to GPU memory).

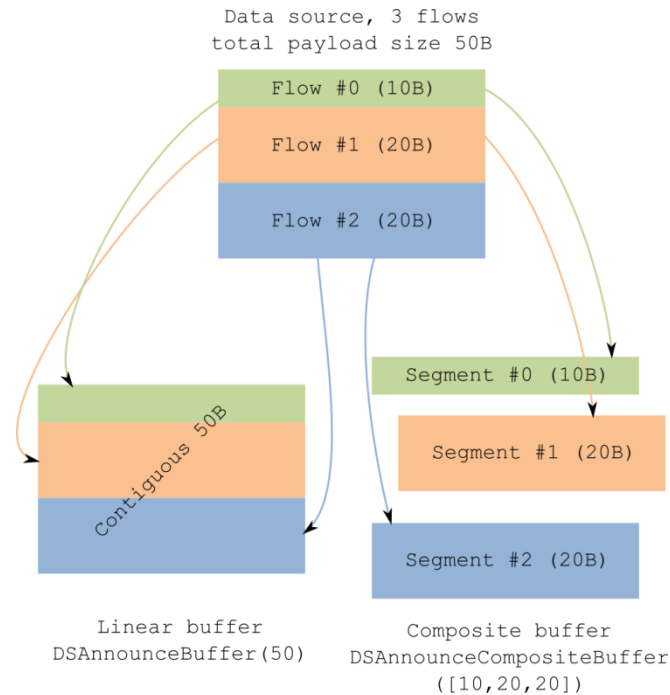
To achieve the first goal (parallel transfer), no special cooperation from GenTL Consumer is required. If acquisition into contiguous buffers is satisfactory, the GenTL Consumer can announce them as usual using [DSAllocAndAnnounceBuffer](#) or [DSAnnounceBuffer](#). The GenTL Producer is responsible to configure the multi-flow acquisition under the hood (according to requirements of its respective transport layer technology) and transfer them into suitable locations of the target buffer. The split of the buffer to per-flow segments happens in this case transparently to GenTL Consumer which receives the data in single buffer, for example as a linear GenDC Container ([5.7.3](#)).

To achieve the second goal (discrete target locations), the GenTL Consumer must announce composite buffers ([DSAnnounceCompositeBuffer](#), see [5.7.1](#)) with structure matching the current flow configuration. In particular, each composite buffer should have not less segments than currently expected flows, each of the segments should not be smaller than the corresponding flow.

When announcing buffers for acquisition, the GenTL Consumer would first query the flow table using [DSGetNumFlows](#) and [DSGetFlowInfo \(FLOW INFO SIZE\)](#). Alternatively, it can query it in GenDC format using [DSGetInfo \(STREAM INFO FLOW TABLE\)](#). Knowing the flow table, it would allocate and announce composite buffers with structure matching the reported flow table, so that each flow could transfer its data to the corresponding buffer segment. In presence of flows, the GenTL Producer must always be able to provide the flow table, otherwise the GenTL Consumer has no way to allocate its composite buffers.

It is important to keep in mind that the flow structure can depend on device configuration and it is thus important to query it after the configuration is finished, just before the acquisition start (refer also to `TLParamsLocked` feature definition in SFNC). When composite buffers are used for acquisition, the flow mapping table defines the buffer allocation requirements in a similar way as `PayloadSize` feature defines it for simple contiguous buffers (as discussed in [5.2.1](#)).

The following figure illustrates acquisition through data stream flows for cases when consumer announces contiguous or composite buffers.



When one or more announced composite buffers are not sufficient for acquisition given the current flow configuration, the [GC_ERR_BUFFER_TOO_SMALL](#) error code is used in the same way as defined for non-composite buffers in [5.2.1](#). It is used the very same way either when starting the acquisition or when discovering the problem at runtime, including use of the [BUFFER_INFO_IS_INCOMPLETE](#) and [BUFFER_INFO_DATA_LARGER_THAN_BUFFER](#) flags.

The configuration of the acquisition data mapping into flows is in general beyond scope of the GenTL specification. The principal use case for flows is, however, GenDC streaming. In case of use with GenDC, the flow mapping rules will adhere to the GenDC specification and related sections in SFNC or GenTL SFNC documents.

5.7.3 GenDC Streaming

GenTL supports data acquisition in the GenDC format ([PAYLOAD_TYPE_GENDC](#)). In this case, the structure and contents of the data is defined by the GenDC specification and should be interpreted according its rules.

To guarantee flawless GenDC data exchange, independent of transport layer specific details, the GenTL Producer and GenTL Consumer must stick to a few basic rules described below.

The buffer delivered with GenDC payload ([PAYLOAD_TYPE_GENDC](#)) must contain exactly one GenDC container descriptor (as defined by GenDC) and it must be the “final” descriptor type. If given transport layer technology does not deliver the descriptor within the data payload (but e.g. within data leader section used by some technologies), the GenTL Producer has to guarantee that the container descriptor is copied into the destination buffer. This implies that the GenTL Producer also provides correct PayloadSize information including space for the descriptor. Note that these requirements are important to hide transport layer

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

specifics and guarantee that acquired buffers are self-described even after device/stream configuration has changed.

The GenDC data can optionally be transferred using stream flows (5.7.2). The GenDC container layout within the buffer memory depends how it was announced – as contiguous or composite buffer.

GenDC container acquired into a contiguous buffer:

- A contiguous buffer is announced using [DSAllocAndAnnounceBuffer](#) or [DSAnnounceBuffer](#) and consists of single blob of contiguous memory.
- The GenTL Producer stores the data into the buffer as a **linear** GenDC container (as defined in the GenDC specification). When using multiple flows, the GenTL Producer is responsible for figuring out the flow destination offsets in the buffer based on the flow mapping table (see 5.7.2).
- The GenDC descriptor is stored at offset 0 from start of the buffer.
- The GenTL Consumer locates the individual GenDC container components/parts' data based on their linear DataOffset's from start of the buffer (DataOffset is a GenDC part header field standardized in GenDC).

GenDC container acquired into a composite buffer:

- A composite buffer is announced using [DSAnnounceCompositeBuffer](#) and consists of multiple memory segments (5.7.1), corresponding to configured data stream flows.
- The GenTL Producer stores the data into the buffer segments „per flow“. Data from flow #0 in segment #0, flow #1 to segment #1, etc. The GenTL Consumer is responsible to query the flow mapping table prior to acquisition start and announce the composite buffers (see 5.7.2).
- The GenDC descriptor is stored at offset 0 from start of the buffer segment #0 (GenDC mandates transferring the descriptor in flow #0).
- The GenTL Consumer locates the individual GenDC container components/parts' data based on their FlowOffset from start of their corresponding buffer segment (FlowOffset is GenDC part header field standardized in GenDC, FlowId field in the same header assigns each part to given flow and thus to corresponding target segment in the composite buffer).

When receiving GenDC payload, the GenTL Consumer must interpret the payload data based on the information stored in the GenDC descriptor rather than querying the data properties (such as image dimensions or pixel format) using [DSGetBufferInfo](#).

The GenTL Producer may support a mode in which it presents all acquired data as GenDC payload, even if it was not originally generated in GenDC format by the device. In this case the GenTL Producer is responsible to provide information about required target buffers (flow table and/or payload size) to accommodate the entire data including the GenDC descriptor.

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 Installation

In order to install a GenTL Producer an installer needs to add the path where the GenTL Producer implementation can be found to a path variable with the name GENICAM_GENTL{32/64}_PATH. The entries within the variable are separated by ‘;’ on Windows and ‘:’ on UNIX based systems. In order to allow different directories for 32Bit and 64Bit implementations residing on the same system two variables are defined: GENICAM_GENTL32_PATH for 32Bit GenTL Producer implementations and GENICAM_GENTL64_PATH for 64Bit GenTL Producer implementations. A consumer may pick the appropriate version of the environment variable.

6.1.2 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 6-3: Function naming convention

Entry	Description
<i>Prefix</i>	<p>Specifies the handle the function works on. The handle represents the module used.</p> <p>Values:</p> <ul style="list-style-type: none"> • 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
<i>Operation</i>	<p>Specifies the operation done on a certain module.</p> <p>Values (choice):</p> <ul style="list-style-type: none"> • Open to instantiate a module

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Entry	Description
	<ul style="list-style-type: none"> • Close to close a module • Get to query information about a module or object
<i>Specifier</i>	<p>The specifier is optional. If an operation needs additional information it is provided by the <i>Specifier</i>.</p> <p>Values (choice):</p> <ul style="list-style-type: none"> • 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.

6.1.3 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 reallocated inside the library. Vice versa all objects and data allocated by the calling GenTL Consumer must only be reallocated and freed by the calling GenTL Consumer. No language specific features except the ones allowed by ANSI C and published in the interface are allowed.

The function names of the exported functions must be undecorated. The function calling convention is `stdcall` for x86 platforms and architecture dependent for other platforms.

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 a second time from within a single process the error [GC_ERR_RESOURCE_IN_USE](#) 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.

6.1.4 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. Certain restrictions apply for all list functions like [TLUpdateInterfaceList](#) and [IFUpdateDeviceList](#) since results are cached inside the module.

If a platform supports independent processes the GenTL Producer implementation may establish interprocess communication. Minimal requirement is that other processes are not allowed to use an opened Device module. It is recommended though that a GenTL Producer implementation is multiprocess capable to the point where:

- Access rights for the Modules are checked
An open Device module should be locked against multiple process write access. In that case an error should be returned. Read access may be granted though.
- 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. Furthermore different processes should be able to communicate with different devices.

6.1.5 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. Therefore the output values of a function are to be handled as if being invalid if the function returns an error code.


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 6-4: C interface error codes

Enumerator	Value	Description
<code>GC_ERR_SUCCESS</code>	0	Operation was successful; no error occurred.
<code>GC_ERR_ERROR</code>	-1001	Unspecified runtime error.
<code>GC_ERR_NOT_INITIALIZED</code>	-1002	Module or resource not initialized; e.g., GCInitLib was not called .
<code>GC_ERR_NOT_IMPLEMENTED</code>	-1003	Requested operation not implemented; e.g., no Port functions on a Buffer module.
<code>GC_ERR_RESOURCE_IN_USE</code>	-1004	Requested resource is already in use.

Enumerator	Value	Description
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 or the data does not provide reliable information corresponding with the request.
GC_ERR_INVALID_PARAMETER	-1009	One of the parameter given was not valid or out of range.
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_ABORT	-1012	An operation has been aborted before it could be completed. For example a wait operation through EventGetData has been terminated via a call to EventKill .
GC_ERR_INVALID_BUFFER	-1013	The GenTL Consumer has not announced enough buffers to start the acquisition in the currently active acquisition mode.
GC_ERR_NOT_AVAILABLE	-1014	Resource or information is not available at a given time in a current state.
GC_ERR_INVALID_ADDRESS	-1015	A given address is out of range or invalid for internal reasons.
GC_ERR_BUFFER_TOO_SMALL	-1016	A provided buffer is too small to receive the expected amount of data. This may affect acquisition buffers in the Data Stream module if the buffers are smaller than the expected payload size but also buffers passed to any other function of the GenTL Producer interface to retrieve information or IDs.
GC_ERR_INVALID_INDEX	-1017	A provided index referencing a

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Enumerator	Value	Description
		Producer internal object is out of bounds.
GC_ERR_PARSING_CHUNK_DATA	-1018	An error occurred parsing a buffer containing chunk data.
GC_ERR_INVALID_VALUE	-1019	A register write function was trying to write an invalid value.
GC_ERR_RESOURCE_EXHAUSTED	-1020	A requested resource is exhausted. This is a rather general error which might for example refer to a limited number of available handles being available.
GC_ERR_OUT_OF_MEMORY	-1021	The system and/or other hardware in the system (frame grabber) ran out of memory.
GC_ERR_BUSY	-1022	The required operation cannot be executed because the responsible module/entity is busy executing actions that cannot be performed concurrently with the requested operation.
GC_ERR_AMBIGUOUS	-1023	The required operation cannot be executed unambiguously in given context.
GC_ERR_CUSTOM_ID	-10000	Any error smaller or equal than -10000 is implementation specific. If a GenTL Consumer receives such an error number it should react as if it would be a generic runtime error.

To get a detailed descriptive text about the error reason call the [GCGetLastError](#) function.

Some error codes might be returned by any function and are therefore not explicitly listed in the function's error code table. These error codes are:

- GC_ERR_ERROR
- GC_ERR_IO
- GC_ERR_RESOURCE_EXHAUSTED
- GC_ERR_OUT_OF_MEMORY

6.1.6 Software Interface Versions

The software interface evolves over the individual versions of the GenTL specification. In particular, between two versions of the interface, new functions (and corresponding data structures) and enumerations might be introduced. In rare cases, existing functions or commands might be conversely deprecated. Interface versions are indicated by a major

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

version number and a minor version number in a notation “x.y” with ‘x’ being the major version number and ‘y’ being the minor version number.

- **Major Version Numbers**
Different major version numbers indicate major additions to the interface and/or breaking changes. This means for example a removal of functions or a complete new feature set. The newer interface is therefore not backward compatible.
- **Minor Version Numbers**
Changes in the minor version number of the software interface may indicate new functionality and clarifications in the text describing the interface. If only the minor version changes the interface stays backward compatible.
Changing feature names without functionl change is also allowed in minor releases.

When developing a GenTL Consumer that should be compatible with a widest range of GenTL Producer versions, special care might be required to consider these differences.

When using an enumeration unknown to the GenTL Producer, the function getting that value as a parameter would return an appropriate error code. For example when querying an unknown info command, the GenTL Producer would return `GC_ERR_NOT_IMPLEMENTED`.

When trying to use a GenTL interface function unknown to the GenTL Producer, the function implementation will be simply missing in the GenTL Producer's binary. For the functions that are not universally available in all GenTL specification versions, the Consumer should check their presence in the GenTL Producer's interface at load time and if possible, consider a suitable fallback behaviour for GenTL Producers not implementing that function.

6.2 Used Data Types

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

6.2.1.1 GC_ERROR

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

6.2.1.2 Handles

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

6.2.1.3 Enumerations

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

6.2.1.4 Buffers and C Strings

Buffers are normally typed as `void*` if arbitrary data is accessed. Specialized buffers like C strings are by default ASCII encoded and a `char*` is used unless reported different through the type information provided by the info functions (for example [IFGetInfo](#)). A `char` is

expected to have 8 bits. On platforms/compilers where this is not the case a byte like primitive data type must be used.

String encoding is by default ASCII (characters with numerical values up to and including 127) unless stated different through the [TL INFO CHAR ENCODING](#) command. A string as an input value without a size parameter must be 0-terminated. Strings with a size parameter must include the terminating 0.

6.2.1.5 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 `ptrdiff_t` is a signed type which indicates that its value relates to an arithmetic operation with a memory pointer, usually a buffer. Its size is platform dependent (e.g., 32 bit on 32 bit platforms and 64Bit on 64Bit 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

6.3.1.1 GCCloseLib

GC ERROR	<code>GCCloseLib</code>	<code>(void)</code>
--------------------------	-------------------------	-----------------------

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. Each call to [GCCloseLib](#) has to be accompanied by a preceding call to [GCInitLib](#).

[GCGetLastError](#) must not be called after the call of this function!

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib or library has already been closed through a call to <code>GCCloseLib</code> .

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.1.2 GCGetInfo

```

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

Inquire information about a GenTL implementation without instantiating a System module. The available information is limited since the TL is not initialized yet. Even if this function works on a library without an instantiated System module, [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] <i>iInfoCmd</i>	Information to be retrieved as defined in TL_INFO_CMD .
[out] <i>piType</i>	Data type of the <i>pBuffer</i> content as defined in the TL_INFO_CMD and INFO_DATATYPE .
[in,out] <i>pBuffer</i>	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>piSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>iType</i> is a string the size includes the terminating 0.
[in,out] <i>piSize</i>	<i>pBuffer</i> equal NULL: out: minimal size of <i>pBuffer</i> in bytes to hold all information <i>pBuffer</i> unequal NULL: in: size of the provided <i>pBuffer</i> in bytes out: number of bytes filled by the function

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.1.3 GCGetLastError

GC_ERROR	GCGetLastError	(GC_ERROR * char * size_t *	<i>piErrorCode</i> , <i>sErrorText</i> , <i>piSize</i>)
--------------------------	----------------	------------------------------------	--

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. In case an error occurs and after that several other function calls return without error the last error value and description is returned and the successful calls are ignored. If there has not been any error in the given thread context since startup the function will return GC_ERR_SUCCESS with **piErrorCode* also set to GC_ERR_SUCCESS and *sErrorText* containing “No Error”. In case GCGetLastError itself generates an error it will return the according error code but it will not store the error internally so that succeeding calls to GCGetLastError will still be able to report the stored error code.

Parameters

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piErrorCode</i> are invalid pointers (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>sErrorText</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.1.4 GCInitLib

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 to [GCInitLib](#) without accompanied calls to [GCCloseLib](#) will return the error GC_ERR_RESOURCE_IN_USE.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_RESOURCE_IN_USE	GCInitLib already called without accompanied call to GCCloseLib .

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2 System Functions

6.3.2.1 TLClose

GC_ERROR	TLClose	(TL_HANDLE <i>hSystem</i>)
--------------------------	---------	------------------------------

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] <i>hSystem</i>	System module handle to close.
---------------------	--------------------------------

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hSystem</i> is invalid (NULL) or does not reference an open System module retrieved through a call to TLOpen .

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.2 TLGetInfo

```
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_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE The handle *hSystem* is invalid (NULL) or does not reference an open System module retrieved through a call to [TLOpen](#).
- GC_ERR_NOT_IMPLEMENTED Specified *iInfoCmd* is not implemented.
- GC_ERR_INVALID_PARAMETER Parameters *piSize* and/or *piType* are invalid pointers (NULL or ~0x0) .
- GC_ERR_BUFFER_TOO_SMALL *pBuffer* is not NULL and the value of **piSize* is too small to receive the expected amount of data.
- GC_ERR_NOT_AVAILABLE The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.3 TLGetInterfaceID

GC_ERROR	TLGetInterfaceID	(TL_HANDLE uint32_t char * size_t *	<i>hSystem</i> , <i>iIndex</i> , <i>sIfaceID</i> , <i>piSize</i>)
--------------------------	------------------	---	---

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.

This function is not thread safe since it relies on an internal cache.

Parameters

[in] <i>hSystem</i>	System module to work on.
[in] <i>iIndex</i>	Zero-based index of the interface on this system.
[in,out] <i>sIfaceID</i>	Pointer to a user allocated C string buffer to receive the Interface module ID at the given <i>iIndex</i> . If this parameter is NULL, <i>piSize</i> will contain the needed size of <i>sIfaceID</i> in bytes. The size includes the terminating 0.
[in,out] <i>piSize</i>	<i>sIfaceID</i> equal NULL: out: minimal size of <i>sIfaceID</i> in bytes to hold all information <i>sIfaceID</i> unequal NULL: in: size of the provided <i>sIfaceID</i> in bytes out: number of bytes filled by the function

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hSystem</i> is invalid (NULL) or does not reference an open System module retrieved through a call to TLOpen .
GC_ERR_INVALID_INDEX	<i>iIndex</i> is greater than the number of available Interface modules - 1 retrieved through a call to TLGetNumInterfaces .
GC_ERR_INVALID_PARAMETER	Parameter <i>piSize</i> is an invalid pointer (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>sIfaceID</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.4 TLGetInterfaceInfo

```

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. The reported information should be in sync to information retrieved through the [IFGetInfo](#) function.

Parameters

[in]	<i>hSystem</i>	System module to work on.
[in]	<i>sIfaceID</i>	Unique ID of the interface to inquire information from. Like with the TLOpenInterface function it is also possible to feed an alternative ID as long as the GenTL Producer knows how to interpret it.
[in]	<i>iInfoCmd</i>	Information to be retrieved as defined in INTERFACE_INFO_CMD .
[out]	<i>piType</i>	Data type of the <i>pBuffer</i> content as defined in the INTERFACE_INFO_CMD and INFO_DATATYPE .
[in,out]	<i>pBuffer</i>	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>piSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>piType</i> is a string the size includes the terminating 0.
[in,out]	<i>piSize</i>	<i>pBuffer</i> equal NULL: out: minimal size of <i>pBuffer</i> in bytes to hold all information <i>pBuffer</i> unequal NULL: in: size of the provided <i>pBuffer</i> in bytes out: number of bytes filled by the function

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hSystem</i> is invalid (NULL) or does not reference an open System module retrieved through a call to TLOpen .
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0).

GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_INVALID_ID	The GenTL Producer is unable to interpret the provided ID string <i>sIfaceID</i> or is not able to match it to an existing Interface.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.5 TLGetNumInterfaces

```
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.

This function is not thread safe since it relies on an internal cache.

Parameters

[in]	<i>hSystem</i>	System module to work on.
[out]	<i>piNumIfaces</i>	Number of interfaces on this System module.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	Either the handle <i>hSystem</i> has an invalid value or the handle does not belong to a previously opened TL module through a call to TLOpen .
GC_ERR_INVALID_PARAMETER	Parameter <i>piNumIfaces</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.6 TLOpen

GC_ERROR TLOpen	(TL_HANDLE * <i>phSystem</i>)
---------------------------------	---------------------------------

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. It is recommended to initialize **phSystem* to [GENTL_INVALID_HANDLE](#) before calling TLOpen to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib
GC_ERR_RESOURCE_IN_USE	The TL module has already been instantiated through a previous call to TLOpen .
GC_ERR_INVALID_PARAMETER	Parameter <i>phSystem</i> is an invalid pointer (NULL or ~0x0).
GC_ERR_ACCESS_DENIED	The access to the requested System module is denied. This may be because it is already opened by another process but it might have other reasons as well.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.7 TLOpenInterface

GC_ERROR TLOpenInterface	(TL_HANDLE <i>hSystem</i> , const char * <i>sIfaceID</i> , IF_HANDLE * <i>phIface</i>)
--	--

Opens the given *sIfaceID* on the given *hSystem*.

Any subsequent call to TLOpenInterface with an *sIfaceID* which has already been opened will return the 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] <i>hSystem</i>	System module to work on.
[in] <i>sIfaceID</i>	Unique interface ID to open as a 0-terminated C string.
[out] <i>phIface</i>	Interface handle of the newly created interface. It is recommended to initialize <i>*phIface</i> to <code>GENTL_INVALID_HANDLE</code> before calling <code>TLOpenInterface</code> to indicate an invalid handle.

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .
<code>GC_ERR_RESOURCE_IN_USE</code>	The Interface module has already been instantiated through a previous call to <code>TLOpenInterface</code> .
<code>GC_ERR_INVALID_HANDLE</code>	The handle <i>hSystem</i> is invalid (NULL) or does not reference an open System module retrieved through a call to TLOpen .
<code>GC_ERR_INVALID_ID</code>	The GenTL Producer is unable to interpret the provided ID string <i>sIfaceID</i> or is not able to match it to an existing Interface.
<code>GC_ERR_INVALID_PARAMETER</code>	Parameters <i>phIface</i> and/or <i>sIfaceID</i> are invalid pointers (NULL or ~0x0).
<code>GC_ERR_ACCESS_DENIED</code>	The access to the requested Interface is denied. This may be because it is already opened by another Process but it might have other reasons as well.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.2.8 TLUpdateInterfaceList

```
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. It is recommended to call `TLUpdateInterfaceList` after reconfiguration of the System module to reflect possible changes.

A call to this function has implications on the thread safety of

- [TLGetNumInterfaces](#)
- [TLGetInterfaceID](#)

Parameters

[in]	<i>hSystem</i>	System module to work on.
[out]	<i>pbChanged</i>	Contains <code>true</code> if the internal list was changed and <code>false</code> otherwise. If set to <code>NULL</code> nothing is written to this parameter.
[in]	<i>iTimeout</i>	Timeout in ms. If set to <code>GENTL_INFINITE</code> the timeout is infinite and the function will only return after the operation is completed. In any case the GenTL Producer must make sure that this operation is completed in a reasonable amount of time depending on the underlying technology. Please be aware that there is no defined way of terminating such an update operation. On the other hand it is the GenTL Consumer's responsibility to call this function with a reasonable timeout.

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib
<code>GC_ERR_INVALID_HANDLE</code>	The handle <i>hSystem</i> is invalid (<code>NULL</code>) or does not reference an open System module retrieved through a call to TLOpen .
<code>GC_ERR_TIMEOUT</code>	The specified <i>iTimeout</i> expired before the Producer was able to completely update the list. In this case the "old" list stays valid.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3 Interface Functions

6.3.3.1 IFClose

GC_ERROR IFClose (IF_HANDLE <i>hIface</i>)
--

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

Parameters

[in]	<i>hIface</i>	Interface module handle to close.
------	---------------	-----------------------------------

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .

GC_ERR_INVALID_HANDLE The handle *hiface* is invalid (NULL) or does not reference an open Interface module retrieved through a call to [TLOpenInterface](#).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.2 IFGetInfo

```

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

Inquires information about the Interface module as defined in [INTERFACE_INFO_CMD](#). The reported information should be in sync to information retrieved through the [TLGetInterfaceInfo](#) function.

Parameters

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib
GC_ERR_INVALID_HANDLE	The handle <i>hiface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.

GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0)
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.3 IFGetDeviceID

```

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.

This function is not thread safe since it relies on an internal cache.

Parameters

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .

GC_ERR_INVALID_INDEX	<i>iIndex</i> is greater than the number of available Device modules - 1 retrieved through a call to IFGetNumDevices .
GC_ERR_INVALID_PARAMETER	Parameter <i>piSize</i> is an invalid pointer (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>sIfaceID</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.4 IFGetDeviceInfo

```

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

Inquires information about a device on the given Interface module *hIface* as defined in [DEVICE_INFO_CMD](#) without the need to open the device. The reported information should be in sync to information returned through the [DevGetInfo](#) function.

Parameters

[in] <i>hIface</i>	Interface module to work on.
[in] <i>sDeviceID</i>	Unique ID of the device to inquire information about. Like with the IFOpenDevice function it is also possible to feed an alternative ID as long as the GenTL Producer knows how to interpret it.
[in] <i>iInfoCmd</i>	Information to be retrieved as defined in DEVICE_INFO_CMD .
[out] <i>piType</i>	Data type of the <i>pBuffer</i> content as defined in the DEVICE_INFO_CMD and INFO_DATATYPE .
[in,out] <i>pBuffer</i>	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>piSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>piType</i> is a string the size includes the terminating 0.
[in,out] <i>piSize</i>	<i>pBuffer</i> equal NULL: out: minimal size of <i>pBuffer</i> in bytes to hold all information <i>pBuffer</i> unequal NULL: in: size of the provided <i>pBuffer</i> in bytes out: number of bytes filled by the function

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .
GC_ERR_INVALID_ID	The GenTL Producer is unable to interpret the provided ID string <i>sDeviceID</i> or is unable to match it to an existing Device.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> and/or <i>sDeviceID</i> are invalid pointers (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.5 IFGetNumDevices

GC ERROR	IFGetNumDevices	(IF_HANDLE	<i>hIface</i> ,
		uint32_t *	<i>piNumDevices</i>)

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.

This function is not thread safe since it relies on an internal cache.

Parameters

[in]	<i>hIface</i>	Interface module to work on.
[out]	<i>piNumDevices</i>	Number of devices on this Interface module.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib
GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .

GC_ERR_INVALID_PARAMETER Parameter *piNumDevices* is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.6 IFOpenDevice

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

Opens the given *sDeviceID* with the given *iOpenFlag* on the given *hIface*.

Any subsequent call to [IFOpenDevice](#) with an *sDeviceID* which has already been opened will return the 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]	<i>hIface</i>	Interface module to work on.
[in]	<i>sDeviceID</i>	Unique device ID to open as a 0-terminated C string.
[in]	<i>iOpenFlag</i>	Configures the open process as defined in the DEVICE ACCESS FLAGS .
[out]	<i>phDevice</i>	Device handle of the newly created Device module. It is recommended to initialize <i>*phDevice</i> to GENTL_INVALID_HANDLE before calling IFOpenDevice to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .
GC_ERR_INVALID_ID	The GenTL Producer is unable to interpret the provided ID string <i>sDeviceID</i> or is not able to match it to an existing Device.
GC_ERR_RESOURCE_IN_USE	The Device module has already been instantiated through a previous call to IFOpenDevice .

GC_ERR_INVALID_PARAMETER	Parameters <i>sDeviceID</i> and/or <i>phDevice</i> are invalid pointers (NULL or ~0x0) or <i>iOpenFlag</i> contains a non valid/unknown value.
GC_ERR_NOT_IMPLEMENTED	<i>iOpenFlag</i> contains a value, which is not implemented by this GenTL Producer.
GC_ERR_ACCESS_DENIED	The access to the requested device is denied.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.7 IFUpdateDeviceList

```
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. It is recommended to call `IFUpdateDeviceList` regularly from time to time and after reconfiguration of the Interface module to reflect possible changes.

A call to this function has implications on the thread safety of

- [IFGetNumDevices](#)
- [IFGetDeviceID](#)

Parameters

[in] <i>hIface</i>	Interface module to work on.
[out] <i>pbChanged</i>	Contains <code>true</code> if the internal list was changed and <code>false</code> otherwise. If set to NULL nothing is written to this parameter.
[in] <i>iTimeout</i>	Timeout in ms. If set to <code>GENTL_INFINITE</code> the timeout is infinite and the function will only return if the operation is completed. In any case the GenTL Producer must make sure that this operation is completed in a reasonable amount of time depending on the underlying technology. Please be aware that there is no defined way of terminating such an update operation. On the other hand it is the GenTL Consumer's responsibility to call this function with a reasonable timeout.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .

GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .
GC_ERR_TIMEOUT	The specified <i>iTimeout</i> expired before the Producer was able to completely update the list. In this case the “old” list stays valid.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.3.8 IFGetParentTL

GC ERROR	IFGetParentTL	(IF_HANDLE	<i>hIface</i> ,
		TL_HANDLE *	<i>phSystem</i>)

Retrieves a handle to the parent TL module.

Parameters

[in]	<i>hIface</i>	Interface module to work on.
[out]	<i>phSystem</i>	Handle to the parent System module

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib
GC_ERR_INVALID_HANDLE	The handle <i>hIface</i> is invalid (NULL) or does not reference an open Interface module retrieved through a call to TLOpenInterface .
GC_ERR_INVALID_PARAMETER	Parameter <i>phSystem</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4 Device Functions

6.3.4.1 DevClose

GC ERROR	DevClose	(DEV_HANDLE	<i>hDevice</i>)
--------------------------	----------	--------------	------------------

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. If DevClose is

called with a handle returned from a call to [DevGetPort](#) a `GC_ERR_INVALID_HANDLE` is to be returned.

Parameters

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

Returns

`GC_ERR_SUCCESS` Operation was successful; no error occurred.

`GC_ERR_NOT_INITIALIZED` No preceding call to [GCInitLib](#).

`GC_ERR_INVALID_HANDLE` The handle *hDevice* is invalid (NULL) or does not reference an open Device module retrieved through a call to [IFOpenDevice](#).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.2 DevGetInfo

GC ERROR	<code>DevGetInfo</code>	(<code>DEV_HANDLE</code> <i>hDevice</i> , <code>DEVICE_INFO_CMD</code> <i>iInfoCmd</i> , <code>INFO_DATATYPE</code> * <i>piType</i> , <code>void</code> * <i>pBuffer</i> , <code>size_t</code> * <i>piSize</i>)
--------------------------	-------------------------	---

Inquire information about the Device module as defined in [DEVICE INFO CMD](#). The reported information should be in sync to information retrieved through the [IFGetDeviceInfo](#) function.

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_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.3 DevGetDataStreamID

```

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.

For GenTL Producers which do not provide a data stream the number of available data streams is zero. Calls to [DevGetDataStreamID](#) or [DevOpenDataStream](#) will fail. Nevertheless a GenTL Producer must export all functions of the public interface.

Parameters

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

in: size of the provided *sDataStreamID* in bytes
out: number of bytes filled by the function

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .
<code>GC_ERR_INVALID_HANDLE</code>	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
<code>GC_ERR_NOT_IMPLEMENTED</code>	The Producer does not implement streaming or the remote device does not provide a stream. DevGetNumDataStreams reports zero.
<code>GC_ERR_INVALID_INDEX</code>	<i>iIndex</i> is greater than the number of available Data Stream modules - 1 retrieved through a call to DevGetNumDataStreams .
<code>GC_ERR_INVALID_PARAMETER</code>	Parameter <i>piSize</i> is an invalid pointer (NULL or ~0x0).
<code>GC_ERR_BUFFER_TOO_SMALL</code>	<i>sDataStreamID</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.4 DevGetNumDataStreams

GC ERROR <code>DevGetNumDataStreams (DEV_HANDLE <i>hDevice</i>, uint32_t * <i>piNumDataStreams</i>)</code>

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

For GenTL Producers which do not provide a data stream the number of available data streams is zero. Calls to [DevGetDataStreamID](#) or [DevOpenDataStream](#) will fail with `GC_ERR_NOT_IMPLEMENTED`. Nevertheless a GenTL Producer must export all functions of the public interface

Parameters

[in] <i>hDevice</i>	Device module to work on.
[out] <i>piNumDataStreams</i>	Number of data stream on this Device module.

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .

GC_ERR_INVALID_HANDLE	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
GC_ERR_INVALID_PARAMETER	Parameter <i>piNumDataStreams</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.5 DevGetPort

GC ERROR	DevGetPort	(DEV_HANDLE	<i>hDevice</i> ,
		PORT_HANDLE *	<i>phRemoteDev</i>)

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.

The remote device Port handle is no valid source for Events. Therefore it must not be used to register Events through [GCRegisterEvent](#).

Parameters

[in] <i>hDevice</i>	Device module to work on.
[out] <i>phRemoteDev</i>	Port handle for the remote device. It is recommended to initialize <i>*phRemoteDev</i> to <code>GENTL_INVALID_HANDLE</code> before calling DevGetPort to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
GC_ERR_INVALID_PARAMETER	Parameter <i>phRemoteDev</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.6 DevOpenDataStream

```
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 the 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.

For GenTL Producers which do not provide a data stream the number of available data streams is zero. Calls to [DevGetDataStreamID](#) or [DevOpenDataStream](#) will fail. Nevertheless a GenTL Producer must export all functions of the public interface.

Parameters

[in] <i>hDevice</i>	Device module to work on
[in] <i>sDataStreamID</i>	Unique data stream ID to open as a 0-terminated C string.
[out] <i>phDataStream</i>	Data Stream module handle of the newly created stream. It is recommended to initialize <i>*phDataStream</i> to GENTL_INVALID_HANDLE before calling DevOpenDataStream to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
GC_ERR_RESOURCE_IN_USE	The Data Stream module has already been instantiated through a previous call to DevOpenDataStream .
GC_ERR_INVALID_ID	The GenTL Producer is unable to interpret the provided ID string <i>sDataStreamID</i> or is not able to match it to an existing Data Stream.
GC_ERR_INVALID_PARAMETER	Parameters <i>phDataStream</i> and/or <i>sDataStreamID</i> are invalid pointers (NULL or ~0x0).

GC_ERR_ACCESS_DENIED	The access to the requested Data Stream module is denied. This may be because it is already opened by another Process but it might have other reasons as well.
GC_ERR_NOT_AVAILABLE	The <i>sDataStreamID</i> of the stream is generally valid but the stream is not available.
GC_ERR_NOT_IMPLEMENTED	The Producer does not implement streaming or the remote device does not provide a stream. DevGetNumDataStreams reports zero.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.4.7 DevGetParentIF

```
GC_ERROR DevGetParentIF ( DEV_HANDLE hDevice,
                          IF_HANDLE * phIface )
```

Retrieves a handle to the parent Interface module.

Parameters

[in] <i>hDevice</i>	Device module to work on.
[out] <i>phIface</i>	Handle to the parent Interface module.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDevice</i> is invalid (NULL) or does not reference an open Device module retrieved through a call to IFOpenDevice .
GC_ERR_INVALID_PARAMETER	Parameter <i>phIface</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5 Data Stream Functions

6.3.5.1 DSAllocAndAnnounceBuffer

```
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 buffer 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 should be matched with a call of [DSRevokeBuffer](#) even though the resources are also freed when the module is closed.

Refer to chapter [5.2.1](#) in order to determine the right buffer size.

Parameters

[in]	<i>hDataStream</i>	Data Stream module to work on.
[in]	<i>iBufferSize</i>	Size of the buffer in bytes.
[in]	<i>pPrivate</i>	Pointer to private data which will be passed to the GenTL Consumer on New Buffer events. This parameter may be NULL.
[out]	<i>phBuffer</i>	Buffer module handle of the newly announced buffer. It is recommended to initialize <i>*phBuffer</i> to GENTL_INVALID_HANDLE before calling DSAllocAndAnnounceBuffer to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_INVALID_PARAMETER	Parameter <i>phBuffer</i> is an invalid pointer (NULL or ~0x0).

GC_ERR_BUSY

The acquisition has been started and the GenTL Producer does not support announcing buffers while the acquisition is active.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.2 DSAnnounceBuffer

```
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 buffer 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 to a given stream. If a GenTL Consumer tries to announce an already announced buffer the function will return the error GC_ERR_RESOURCE_IN_USE. A buffer may additionally be announced to one or more other data stream(s) which will then result in one or more additional handles. The Consumer needs to take care about synchronisation between these streams.

Refer to chapter [5.2.1](#) in order to determine the right buffer size.

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 `New Buffer` events. This parameter may be NULL.
- [out] *phBuffer* Buffer module handle of the newly announced buffer. It is recommended to initialize **phBuffer* to `GENTL_INVALID_HANDLE` before calling [DSAnnounceBuffer](#) to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_INVALID_PARAMETER	Parameters <i>pBuffer</i> and/or <i>phBuffer</i> are invalid pointers (NULL or ~0x0).
GC_ERR_RESOURCE_IN_USE	The specified <i>pBuffer</i> is already announced to this Data Stream module or, depending on the implementation of the GenTL Producer, it has already been announced to another instance of the Data Stream module (see chapter 3.6).
GC_ERR_BUSY	The acquisition has been started and the GenTL Producer does not support announcing buffers while the acquisition is active.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.3 DSAnnounceCompositeBuffer

```
GC ERROR DSAnnounceCompositeBuffer ( DS_HANDLE  hDataStream,
                                     size_t      iNumSegments
                                     void **      ppSegments,
                                     size_t *     piSizes,
                                     void *       pPrivate,
                                     BUFFER_HANDLE * phBuffer )
```

This announces GenTL Consumer allocated memory to the Data Stream associated with the *hDataStream* handle and returns a buffer handle which references that composite buffer until the buffer is revoked. This will allocate internal resources which will be freed upon a call to [DSRevokeBuffer](#).

In contrast to [DSAnnounceBuffer](#), this function allows to announce a buffer not referring a single contiguous block of memory, but consisting of multiple segments. This allows routing logically independent portions of the acquired data to separate memory locations, for example when streaming data over multiple flows mechanism. The segments may correspond with different memory areas as well as form a contiguous block, but must not overlap.

All the segments of the announced composite buffer are treated as a single entity by all functions operating on a buffer handle.

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 *ppSegments* must stay valid until the buffer is revoked with [DSRevokeBuffer](#). Every call of this function must be matched with a call of [DSRevokeBuffer](#).

A composite buffer (and any of its segments) can only be announced once to a given stream. If a GenTL Consumer tries to announce an already announced buffer the function will return the error GC_ERR_RESOURCE_IN_USE. A buffer (with its segments) may additionally be announced to one or more other data stream(s) which will then result in one or more additional handles. The GenTL Consumer needs to take care about synchronisation between these streams.

Refer to chapter [5.7](#) in order to determine the buffer structure.

Note: there is no alloc-and-announce version of this function.

Parameters

[in]	<i>hDataStream</i>	Data Stream module to work on.
[in]	<i>iNumSegments</i>	Number of segments constituting the composite buffer.
[in]	<i>ppSegments</i>	Pointers to memory of individual segments of the composite buffer to announce. The array must contain <i>iNumSegments</i> items.
[in]	<i>piSizes</i>	Size of the segments in bytes. The array must contain <i>iNumSegments</i> items, one for each segment in <i>ppSegments</i> .
[in]	<i>pPrivate</i>	Pointer to private data which will be passed to the GenTL Consumer on <code>New Buffer</code> events. This parameter may be NULL.
[out]	<i>phBuffer</i>	Buffer module handle of the newly announced composite buffer. It is recommended to initialize <i>*phBuffer</i> to <code>GENTL_INVALID_HANDLE</code> before calling DSAnnounceCompositeBuffer to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_NOT_IMPLEMENTED:	The GenTL implementation does not support composite buffers.
GC_ERR_INVALID_PARAMETER	Parameters <i>ppSegments</i> and/or <i>piSizes</i> and/or <i>phBuffer</i> are invalid pointers or any of the segment pointers in

the *ppSegments* array is invalid pointer (NULL or ~0x0) or *iNumSegments* is zero.

GC_ERR_RESOURCE_IN_USE

One or more of the specified *ppSegments* is already announced to this Data Stream module or, depending on the implementation of the GenTL Producer, it has already been announced to another instance of the Data Stream module (see chapter 3.6).

GC_ERR_BUSY

The acquisition has been started and the GenTL Producer does not support announcing buffers while the acquisition is active.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.4 DSClose

GC ERROR	DSClose	(DS_HANDLE	<i>hDataStream</i>)
--------------------------	---------	-------------	----------------------

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_ERR_SUCCESS

Operation was successful; no error occurred.

GC_ERR_NOT_INITIALIZED

No preceding call to [GCInitLib](#).

GC_ERR_INVALID_HANDLE

The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.5 DSFlushQueue

GC ERROR	DSFlushQueue	(DS_HANDLE	<i>hDataStream</i> ,
		ACQ_QUEUE_TYPE	<i>iOperation</i>)

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_ERR_SUCCESS Operation was successful; no error occurred.
 GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
 GC_ERR_INVALID_HANDLE The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#).
 GC_ERR_NOT_IMPLEMENTED *iOperation* is not implemented.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.6 DSGetBufferID

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

DSGetBufferID queries the buffer handle for a given index *iIndex*. The buffer handle *phBuffer* works as a unique ID of an instance of the Buffer module. The relation between an index *iIndex* and a particular buffer stays valid until a buffer revoked. The index reflects the order in which buffers are announced. If new buffers are announced they are to be appended at the end. If buffers “in the middle” are revoked the sequentially following buffers move into that position. The index stays continuous. So for example if you have 10 buffers announced and you remove the buffer with the id of index 5 you still have the index range from 0 to 8.

Note that the relation between index and buffer handle might change with revoked buffers. As long as no buffers are 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_ERR_SUCCESS Operation was successful; no error occurred.
 GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).

GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_INVALID_INDEX	<i>iIndex</i> is greater than the number of announced buffers through calls to one of the buffer announcement functions.
GC_ERR_INVALID_PARAMETER	Parameter <i>phBuffer</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.7 DSGetBufferInfo

```
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](#).

To retrieve multiple infos about a buffer at once and reduce the number of calls from the GenTL Consumer to the GenTL Producer, [DSGetBufferInfoStacked](#) function can be used instead.

Parameters

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream or the handle <i>hBuffer</i> is invalid (NULL) or does not reference an announced Buffer.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0)
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NO_DATA:	The buffer referenced by <i>hBuffer</i> contains structured data (GenDC or multi-part payload) and given <i>iInfoCmd</i> is not available at global buffer level (refer to BUFFER INFO CMD).
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.8 DSGetInfo

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

Inquires information about the Data Stream module associated with *hDataStream* as defined in [STREAM INFO CMD](#).

Parameters

[in]	<i>hDataStream</i>	Data Stream module to work on.
[in]	<i>iInfoCmd</i>	Information to be retrieved as defined in STREAM INFO CMD .
[out]	<i>piType</i>	Data type of the <i>pBuffer</i> 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_ERR_SUCCESS Operation was successful; no error occurred.

GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).

GC_ERR_INVALID_HANDLE The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#).

GC_ERR_NOT_IMPLEMENTED Specified *iInfoCmd* is not implemented.

GC_ERR_INVALID_PARAMETER Parameters *piSize* and/or *piType* are invalid pointers (NULL or ~0x0)

GC_ERR_BUFFER_TOO_SMALL *pBuffer* is not NULL and the value of **piSize* is too small to receive the expected amount of data.

GC_ERR_NOT_AVAILABLE The request is implemented but the requested information is currently not available for any reason.

GC_ERR_NO_DATA The request is implemented but the requested query is not applicable in current configuration (refer to guidelines provided with individual info commands).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.9 DSQueueBuffer

```
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) if it is not currently queued. Furthermore, a buffer which is already waiting to be delivered cannot be queued for acquisition. A queued buffer cannot 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_ERR_SUCCESS Operation was successful; no error occurred.
 GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
 GC_ERR_INVALID_HANDLE The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or *hBuffer* is invalid (NULL) or does not reference an announced Buffer.
 GC_ERR_RESOURCE_IN_USE The buffer is already queued.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.10 DSRevokeBuffer

```
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. It is up to the implementation/technology if the buffer can be revoked during an ongoing acquisition if it is not queued.

Note that the *ppBuffer* parameter must be used with care, since it is intended only for use with buffers announced using [DSAnnounceBuffer](#), in all other cases it returns NULL. In particular for buffers announced using [DSAnnounceCompositeBuffer](#) the GenTL Consumer has to keep track of the associated segments and their allocated memory itself. It might be therefore suitable if it keeps track of the buffer memory resources in all cases.

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 contiguous memory ([DSAnnounceBuffer](#)) is used which is to be freed. If the buffer was allocated by the GenTL Producer ([DSAllocAndAnnounceBuffer](#)) or if the buffer is a

[out] *ppPrivate* composite buffer ([DSAnnounceCompositeBuffer](#))
 NULL is to be returned. If the parameter is set to NULL it is ignored.
 Pointer to the user data pointer given in the buffer announcement function. If the parameter is set to NULL it is ignored.

Returns

`GC_ERR_SUCCESS` Operation was successful; no error occurred.

`GC_ERR_NOT_INITIALIZED` No preceding call to [GCInitLib](#).

`GC_ERR_INVALID_HANDLE` The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or *hBuffer* is invalid (NULL) or does not reference an announced Buffer.

`GC_ERR_BUSY` The buffer is currently queued and can not be revoked or the GenTL Consumer tried to revoke the buffer while the acquisition was in progress and the implementation or the underlying technology would not allow it.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.11 DSStartAcquisition

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

Starts the acquisition engine on the host. Each call to [DSStartAcquisition](#) must be accompanied by a call to [DSStopAcquisition](#).

Parameters

[in] *hDataStream* Data Stream module to work on.

[in] *iStartFlags* As defined in [ACQ_START_FLAGS](#).

[in] *iNumToAcquire* Sets the number of filled/delivered buffers after which the acquisition engine stops automatically. Buffers which are internally discarded or missed are not counted. If set to [GENTL_INFINITE](#) the acquisition continues until a call to [DSStopAcquisition](#) is issued. If set to 0 a `GC_ERR_INVALID_PARAMETER` is returned.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_NOT_IMPLEMENTED	One or more flags set in <i>iStartFlags</i> referencing functionality which is not implemented.
GC_ERR_INVALID_PARAMETER	<i>iNumToAcquire</i> is 0.
GC_ERR_INVALID_BUFFER	The number of buffers announced through one of the buffer announcement functions is smaller than the number retrieved through a call to DSGetInfo using the STREAM INFO BUFF ANNOUNCE MIN command.
GC_ERR_RESOURCE_IN_USE	The Acquisition is already active.
GC_ERR_BUFFER_TOO_SMALL	One or more of the announced buffers are smaller than the expected payload size required. This is optional to the GenTL Producer implementation if it chooses to not start acquisition in this case or if the acquisition is started and the buffers are not or only partially filled (see chapter 5.2.1).

Error cases not covered in the list above may return error codes according to chapter [6.1.5](#) Error Handling on page [61](#).

6.3.5.12 DSStopAcquisition

```
GC ERROR DSStopAcquisition ( DS_HANDLE hDataStream,
                             ACQ_STOP_FLAGS iStopFlags )
```

Stops the acquisition engine on the host. There must be a call to [DSStopAcquisition](#) accompanying each call to [DSStartAcquisition](#) even though the stream already stopped because the number of frames to acquire was reached. This is also independent of the acquisition modes.

Parameters

- [in] *hDataStream* Data Stream module to work on.
- [in] *iStopFlags* Stops the acquisition as defined in [ACQ_STOP_FLAGS](#).

Returns

- GC_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).

GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_NOT_IMPLEMENTED	One or more flags set in <i>iStopFlags</i> referencing functionality which is not implemented.
GC_ERR_RESOURCE_IN_USE	The Acquisition has already been terminated or it has not been started.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.13 DSGetBufferChunkData

```
GC ERROR DSGetBufferChunkData ( DS_HANDLE      hDataStream,
                                BUFFER_HANDLE  hBuffer,
                                SINGLE_CHUNK_DATA * pChunkData,
                                size_t *      piNumChunks )
```

DSGetBufferChunkData parses the transport layer technology dependent chunk data info in the buffer. The layout of the chunk data present in the buffer is returned in the *pChunkData* array, one entry per chunk. Every single chunk is described using its ChunkID, offset in the buffer and chunk data size.

Note that for composite buffers, the individual chunk offsets reported in *pChunkData* are linear offsets within entire payload (see also [5.4.1.1](#)).

When dealing with buffers containing standard self-described containers (such as GenDC), the GenTL Consumer must parse and interpret the chunk data directly according to the given container type specification without using this function.

Parameters

[in] <i>hDataStream</i>	Data Stream module to work on.
[in] <i>hBuffer</i>	Buffer handle to parse.
[out] <i>pChunkData</i>	GenTL Consumer allocated array of structures to receive the chunk layout information. If this parameter is NULL, <i>piNumChunks</i> will contain the number of chunks in the buffer, e.g., the minimal number of entries in the <i>pChunkData</i> array.
[in,out] <i>piNumChunks</i>	<i>pChunkData</i> equal NULL: out: number of chunks in the buffer (minimal number of entries in the <i>pChunkData</i> array to hold all information). <i>pChunkData</i> unequal NULL: in: number of entries in the provided <i>pChunkData</i> array. out: number of entries successfully written to the <i>pChunkData</i> array.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream or the handle <i>hBuffer</i> is invalid (NULL) or does not reference an announced Buffer.
GC_ERR_INVALID_PARAMETER	Parameter <i>piNumChunks</i> is an invalid pointer (NULL or ~0x0)
GC_ERR_NO_DATA	The Buffer referenced by <i>hBuffer</i> does not contain chunk data or the buffer contains a standard self-described container (such as GenDC), i.e. the chunk parsing is expected to be handled by the GenTL Consumer.
GC_ERR_BUFFER_TOO_SMALL	<i>pChunkData</i> is not NULL and the value of <i>*piNumChunks</i> is too small to receive the expected amount of data.
GC_ERR_PARSING_CHUNK_DATA	An error occurred during the parsing of the chunk buffer.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.14 DSGetParentDev

GC ERROR	DSGetParentDev	(DS_HANDLE DEV_HANDLE *	<i>hDataStream</i> , <i>phDevice</i>)
--------------------------	----------------	-----------------------------	---

Retrieves a handle to the parent Device module.

Parameters

[in]	<i>hDataStream</i>	Data Stream module to work on.
[out]	<i>phDevice</i>	Handle to the parent Device module.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .

- GC_ERR_INVALID_HANDLE The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#).
- GC_ERR_INVALID_PARAMETER Parameter *phDevice* is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.15 DSGetNumBufferParts

```
GC ERROR DSGetNumBufferParts ( DS_HANDLE            hDataStream,
                               BUFFER_HANDLE        hBuffer,
                               uint32_t *            piNumParts )
```

Inquires the number of independent data parts in the buffer. The GenTL Producer may return 0 in **piNumParts* in case the buffer payload is multipart but does not contain any parts. For example in case the individual parts of a multipart buffer can be enabled or disabled in the camera it can happen that a camera sends a multipart payload with no parts enabled and maybe only chunk data is being delivered. As described in this case **piNumParts* would report 0. Detailed information about the individual parts can be queried using function [DSGetBufferPartInfo](#).

If the buffer content can be fully described using the information available through [DSGetBufferInfo](#) queries, it is not split into parts and the buffer payload is not multi-part the GenTL Producer must return the error [GC_ERR_NO_DATA](#). The GenTL Consumer would in this case avoid querying information about buffer parts using [DSGetBufferPartInfo](#).

If the reported payload is multi-part the GenTL Producer must use [DSGetNumBufferParts](#) and [DSGetBufferPartInfo](#) to provide information about the buffer.

Parameters

- [in] *hDataStream* Data Stream module to work on.
- [in] *hBuffer* Buffer handle to retrieve information about.
- [out] *piNumParts* Number of independent data parts in the buffer. The reported number may be 0 in case the referenced buffer carries a multipart buffer payload but for some reason the number of parts is buffer is 0.

Returns

- GC_ERR_SUCCESS: Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED: No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE: The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through

a call to [DevOpenDataStream](#) or the handle *hBuffer* is invalid (NULL) or does not reference an announced Buffer.

GC_ERR_NOT_IMPLEMENTED: The GenTL implementation does not support querying information about buffer parts.

GC_ERR_INVALID_PARAMETER: Parameter *piNumParts* is an invalid pointer (NULL or ~0x0).

GC_ERR_NO_DATA: The GenTL implementation supports querying information about buffer parts, but the information about number of data parts in the buffer is currently not available for any reason, for example because the buffer does not contain multi-part payload.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.16 DSGetBufferPartInfo

```
GC ERROR DSGetBufferPartInfo ( DS_HANDLE      hDataStream,
                               BUFFER_HANDLE  hBuffer,
                               uint32_t      iPartIndex,
                               BUFFER_PART_INFO_CMD iInfoCmd,
                               INFO_DATATYPE * piType,
                               void *        pBuffer,
                               size_t *      piSize )
```

Inquires information about individual data parts of the buffer encapsulated in the Buffer module associated with *hBuffer* on the *hDataStream* instance as defined in [BUFFER_PART_INFO_CMD](#).

To retrieve multiple infos about one or more buffer parts at once and reduce the number of calls from the GenTL Consumer to the GenTL Producer, [DSGetBufferPartInfoStacked](#) function can be used instead.

Parameters

- [in] *hDataStream* Data Stream module to work on.
- [in] *hBuffer* Buffer handle to retrieve information about.
- [in] *iPartIndex* Zero based index of the buffer part to query.
- [in] *iInfoCmd* Information to be retrieved as defined in [BUFFER_PART_INFO_CMD](#).
- [out] *piType* Data type of the *pBuffer* content as defined in the [BUFFER_PART_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

[in,out] *piSize*

minimal size of *pBuffer* in bytes. If the *piType* is a string the size includes the terminating 0.

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_ERR_SUCCESS: Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED: No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE: The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or the handle *hBuffer* is invalid (NULL) or does not reference an announced Buffer.
- GC_ERR_NOT_IMPLEMENTED: Specified *iInfoCmd* is not implemented or the GenTL implementation does not support querying information about buffer parts.
- GC_ERR_INVALID_PARAMETER: Parameters *piSize* and/or *piType* are invalid pointers (NULL or ~0x0).
- GC_ERR_INVALID_INDEX: *iPartIndex* is greater than the number of available buffer parts - 1 retrieved through a call to [DSGetNumBufferParts](#).
- GC_ERR_NO_DATA: The buffer referenced by *hBuffer* does not contain data parts.
- GC_ERR_BUFFER_TOO_SMALL_: *pBuffer* is not NULL and the value of **piSize* is too small to receive the expected amount of data.
- GC_ERR_NOT_AVAILABLE: The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.17 DSGetBufferInfoStacked

```
GC_ERROR DSGetBufferInfoStacked ( DS_HANDLE      hDataStream,
                                  BUFFER_HANDLE  hBuffer,
                                  DS_BUFFER_INFO_STACKED *
                                  pInfoStacked,
                                  size_t         iNumInfos )
```

Inquire various information about the Buffer module associated with *hBuffer* on the *hDataStream* instance as defined in [BUFFER_INFO_CMD](#).

With this function, multiple pieces of information can be queried through a single call to the GenTL Producer without the need to combine that information into a custom structure.

Each buffer info is grouped in a structure as defined in [DS_BUFFER_INFO_STACKED](#). A pointer to an array of one or more of these structures is used as in and out parameter. Each structure [DS_BUFFER_INFO_STACKED](#) of that array passes a [BUFFER_INFO_CMD](#) as input and retrieves the required info as output. The details of handling the data members of the [DS_BUFFER_INFO_STACKED](#) structure are defined in [6.5.1.2](#).

Note that the results of the individual queries requested in *pInfoStacked* do not affect return value of the function, nor the last error information reported by [GCGetLastError](#). Even if certain individual queries fail (for example if given info is not available), the function attempts to process all required infos and reports success, unless the call fails as a whole.

Parameters

[in] <i>hDataStream</i>	Data Stream module to work on.
[in] <i>hBuffer</i>	Buffer handle to retrieve information about.
[in,out] <i>pInfoStacked</i>	User allocated array of structures as defined in DS_BUFFER_INFO_STACKED to receive the requested information. Its length is defined by <i>iNumInfos</i> . The array contains the various information to be retrieved as defined in BUFFER_INFO_CMD , on output it provides the results. The details of the structure members and their use for the info exchange are defined in DS_BUFFER_INFO_STACKED (6.5.1.2) .
[in] <i>iNumInfos</i>	Number of stacked buffer infos to retrieve.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred. This does not indicate whether every buffer info was retrieved successfully. Therefore each <i>DS_BUFFER_INFO_STACKED::iResult</i> needs to be checked individually as described in DS_BUFFER_INFO_STACKED .
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .

- GC_ERR_INVALID_HANDLE The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or the handle *hBuffer* is invalid (NULL) or does not reference an announced Buffer module.

- GC_ERR_INVALID_PARAMETER Parameter *pInfoStacked* is an invalid pointer (NULL or ~0x0) or *iNumInfos* is 0.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.18 DSGetBufferPartInfoStacked

```
GC_ERROR DSGetBufferPartInfoStacked ( DS_HANDLE hDataStream,
                                     BUFFER_HANDLE hBuffer,
                                     DS_BUFFER_PART_INFO_STACKED *
                                     pInfoStacked,
                                     size_t iNumInfos )
```

Inquires various information about individual data parts of the buffer encapsulated in the Buffer module associated with *hBuffer* on the *hDataStream* instance as defined in [BUFFER PART INFO CMD](#).

With this function, multiple pieces of information can be queried through a single call to the GenTL Producer without the need to combine that information into a custom structure.

Each buffer part info is grouped in a structure as defined in [DS BUFFER PART INFO STACKED](#). A pointer to an array of one or more of these structures is used as in and out parameter. Each structure [DS BUFFER PART INFO STACKED](#) of that array passes part index and a [BUFFER PART INFO CMD](#) as input and retrieves the required info as output. The details of handling the data members of the [DS BUFFER PART INFO STACKED](#) structure are defined in [6.5.1.3](#).

Note that the results of the individual queries requested in *pInfoStacked* do not affect return value of the function, nor the last error information reported by [GCGetLastError](#). Even if certain individual queries fail (for example if given info is not available), the function attempts to process all required infos and reports success, unless the call fails as a whole.

Parameters

- [in] *hDataStream* Data Stream module to work on.
- [in] *hBuffer* Buffer handle to retrieve information about.
- [in,out] *pInfoStacked* User allocated array of structures as defined in [DS BUFFER PART INFO STACKED](#) to receive the requested information. Its length is defined by *iNumInfos*. The array contains the various information to be retrieved as defined in [BUFFER PART INFO CMD](#), on output it

provides the results. The details of the structure members and their use for the info exchange are defined in [DS_BUFFER_PART_INFO_STACKED \(6.5.1.3\)](#).

[in] *iNumInfos*

Number of stacked buffer part infos to retrieve.

Returns

- GC_ERR_SUCCESS: Operation was successful; no error occurred. This does not indicate whether every buffer part info was retrieved successfully. Therefore each *DS_BUFFER_PART_INFO_STACKED::iResult* needs to be checked individually as described in [DS_BUFFER_PART_INFO_STACKED](#).
- GC_ERR_NOT_INITIALIZED: No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE: The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or the handle *hBuffer* is invalid (NULL) or does not reference an announced Buffer module.
- GC_ERR_NOT_IMPLEMENTED: The GenTL implementation does not support querying information about buffer parts.
- GC_ERR_INVALID_PARAMETER: Parameter *pInfoStacked* is an invalid pointer (NULL or ~0x0) or *iNumInfos* is 0.
- GC_ERR_NO_DATA: The buffer referenced by *hBuffer* does not contain data parts.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.19 DSGetNumFlows

GC ERROR DSGetNumFlows (DS_HANDLE <i>hDataStream</i> , uint32_t * <i>piNumFlows</i>)

Inquires the number of flows currently configured for this data stream.

Note that the number of flows and their structure can be dynamic and depend on the configuration of the data stream and of the device output. It is recommended to query it after any configuration possibly affecting the stream and its contents is finished.

Note also that in case of GenDC streaming the number of flows corresponds to the size of the GenDC flow mapping table.

If querying flow-related information is not relevant in the moment, for example because the device itself does not support flows (or does not expose them in current working mode), the

function must return the error [GC_ERR_NO_DATA](#). The GenTL Consumer would in this case avoid querying information about flows using [DSGetFlowInfo](#).

Parameters

[in] *hDataStream* Data Stream module to work on.
[out] *piNumFlows* Number of flows currently configured for this data stream.

Returns

GC_ERR_SUCCESS: Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED: No preceding call to [GCInitLib](#).
GC_ERR_INVALID_HANDLE: The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#).
GC_ERR_NOT_IMPLEMENTED: The GenTL implementation does not support the flow functionality.
GC_ERR_INVALID_PARAMETER: Parameter *piNumFlows* is an invalid pointer (NULL or ~0x0).
GC_ERR_NO_DATA: The GenTL implementation supports flows, but the flow functionality is currently not available for any reason, for example because the given device/stream does not support flows or is in mode not involving flows.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.20 DSGetFlowInfo

GC ERROR	DSGetFlowInfo	(DS_HANDLE uint32_t FLOW_INFO_CMD INFO_DATATYPE * void * size_t *	<i>hDataStream</i> , <i>iFlowIndex</i> , <i>iInfoCmd</i> , <i>piType</i> , <i>pBuffer</i> , <i>piSize</i>)
--------------------------	---------------	---	--

Inquires information about individual flows currently configured for this data stream as defined in [FLOW_INFO_CMD](#).

Note that the number of flows and their structure can be dynamic and depend on the configuration of the data stream and of the device output. It is recommended to query it after any configuration possibly affecting the stream and its contents is finished.

Parameters

[in] *hDataStream* Data Stream module to work on.

[in]	<i>iFlowIndex</i>	Zero based index of the flow to query.
[in]	<i>iInfoCmd</i>	Information to be retrieved as defined in FLOW_INFO_CMD .
[out]	<i>piType</i>	Data type of the <i>pBuffer</i> content as defined in the FLOW_INFO_CMD and INFO_DATATYPE .
[in,out]	<i>pBuffer</i>	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>piSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>piType</i> is a string the size includes the terminating 0.
[in,out]	<i>piSize</i>	<p><i>pBuffer</i> equal NULL :</p> <p>out: minimal size of <i>pBuffer</i> in bytes to hold all information.</p> <p><i>pBuffer</i> unequal NULL :</p> <p>in: size of the provided <i>pBuffer</i> in bytes.</p> <p>out: number of bytes filled by the function.</p>

Returns

GC_ERR_SUCCESS:	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED:	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE:	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream .
GC_ERR_NOT_IMPLEMENTED:	Specified <i>iInfoCmd</i> is not implemented or the GenTL implementation does not support querying information about flows.
GC_ERR_INVALID_PARAMETER:	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0).
GC_ERR_INVALID_INDEX:	<i>iFlowIndex</i> is greater than the number of available flows - 1 retrieved through a call to DSGetNumFlows .
GC_ERR_NO_DATA:	The data stream referenced by <i>hDataStream</i> has currently no configured flows. This applies also if the actual device does not support flows or is in mode not involving flows
GC_ERR_BUFFER_TOO_SMALL_:	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE:	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.21 DSGetNumBufferSegments

GC_ERROR	DSGetNumBufferSegments (DS_HANDLE <i>hDataStream</i> , BUFFER_HANDLE <i>hBuffer</i> , uint32_t * <i>piNumSegments</i>)
--------------------------	--

Inquires the number of segments in a buffer.

If *hBuffer* references a composite buffer (announced using [DSAnnounceCompositeBuffer](#)), the function reports the number of “real” segments, as announced in [DSAnnounceCompositeBuffer](#).

If *hBuffer* references a contiguous buffer (announced using [DSAllocAndAnnounceBuffer](#) or [DSAnnounceBuffer](#)), the function reports the number of “virtual” segments created within the buffer by the acquisition engine when it was last time filled. When flows were used for the acquisition, the virtual segments correspond to the flow structure ([5.7.2](#)), otherwise the buffer is assumed to consist of a single segment.

Parameters

[in] <i>hDataStream</i>	Data Stream module to work on.
[in] <i>hBuffer</i>	Buffer handle to retrieve information about.
[out] <i>piNumSegments</i>	Number of segments the composite buffer contains. The reported number is expected to be non-zero.

Returns

GC_ERR_SUCCESS:	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED:	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE:	The handle <i>hDataStream</i> is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to DevOpenDataStream or the handle <i>hBuffer</i> is invalid (NULL) or does not reference an announced Buffer.
GC_ERR_NOT_IMPLEMENTED:	The GenTL implementation does not support composite buffers.
GC_ERR_INVALID_PARAMETER:	Parameter <i>piNumSegments</i> is an invalid pointer (NULL or ~0x0).

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.5.22 DSGetBufferSegmentInfo

```
GC_ERROR DSGetBufferSegmentInfo ( DS_HANDLE    hDataStream,
                                   BUFFER_HANDLE hBuffer,
                                   uint32_t      iSegmentIndex,
                                   SEGMENT_INFO_CMD iInfoCmd,
                                   INFO_DATATYPE * piType,
                                   void *        pBuffer,
                                   size_t *      piSize )
```

Inquires information about individual segments of a buffer as defined in [SEGMENT INFO CMD](#).

If *hBuffer* references a composite buffer (announced using [DSAnnounceCompositeBuffer](#)), the function reports information about “real” segments, as announced in [DSAnnounceCompositeBuffer](#).

If *hBuffer* references a contiguous buffer (announced using [DSAllocAndAnnounceBuffer](#) or [DSAnnounceBuffer](#)), the function reports information about virtual segments created within the buffer by the acquisition engine when it was last time filled. When flows were used for the acquisition, the virtual segments correspond to the flow structure ([5.7.2](#)), otherwise the buffer is assumed to consist of a single segment.

Parameters

- [in] *hDataStream* Data Stream module to work on.
- [in] *hBuffer* Buffer handle to retrieve information about.
- [in] *iSegmentIndex* Zero based index of the buffer segment to query.
- [in] *iInfoCmd* Information to be retrieved as defined in [SEGMENT INFO CMD](#).
- [out] *piType* Data type of the *pBuffer* content as defined in the [SEGMENT 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_ERR_SUCCESS: Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED: No preceding call to [GCInitLib](#).

- GC_ERR_INVALID_HANDLE:** The handle *hDataStream* is invalid (NULL) or does not reference an open Data Stream module retrieved through a call to [DevOpenDataStream](#) or the handle *hBuffer* is invalid (NULL) or does not reference an announced Buffer.
- GC_ERR_NOT_IMPLEMENTED:** Specified *iInfoCmd* is not implemented or the GenTL implementation does not support composite buffers.
- GC_ERR_INVALID_PARAMETER:** Parameters *piSize* and/or *piType* are invalid pointers (NULL or ~0x0).
- GC_ERR_INVALID_INDEX:** *iSegmentIndex* is greater than the number of buffer segments - 1 retrieved through a call to [DSGetNumBufferSegments](#).
- GC_ERR_BUFFER_TOO_SMALL:** *pBuffer* is not NULL and the value of **piSize* is too small to receive the expected amount of data.
- GC_ERR_NOT_AVAILABLE:** The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6 Port Functions

6.3.6.1 GCGetPortInfo

GC ERROR	GCGetPortInfo	(PORT_HANDLE	<i>hPort</i> ,
			PORT_INFO_CMD	<i>iInfoCmd</i> ,
			INFO_DATATYPE *	<i>piType</i> ,
			void *	<i>pBuffer</i> ,
			size_t *	<i>piSize</i>)

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_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented or the provided module handle does not have a Port module implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0)
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.
GC_ERR_IO	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.2 GCGetPortURL

GC ERROR	GCGetPortURL	(PORT_HANDLE	<i>hPort</i> ,
			char *	<i>sURL</i> ,
			size_t *	<i>piSize</i>)

GCGetPortURL retrieves a URL list with the XML description for the given *hPort*. See [4.1.2 XML Description](#) page 32 for more information about supported URLs. Each URL is terminated with a trailing '\0' and after the last URL are two '\0'.

In case of multiple XMLs in the device the [GCGetNumPortURLs](#) and [GCGetPortURLInfo](#) should be used.

This function has been deprecated. Producers should support the new functions [GCGetNumPortURLs](#) and [GCGetPortURLInfo](#). In this case this function may only return a subset of the available URLs in the string list. It is up to the implementor which URL to return.

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_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE The handle *hPort* is invalid (NULL) or does not reference an open module.
- GC_ERR_BUFFER_TOO_SMALL *sURL* is not NULL and the value of **piSize* is too small to receive the expected amount of data.
- GC_ERR_INVALID_PARAMETER Parameter *piSize* is an invalid pointer (NULL or ~0x0).
- GC_ERR_NOT_IMPLEMENTED The provided module handle does not have a Port module implemented.
- GC_ERR_IO Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.3 GCGetNumPortURLs

```
GC_ERROR GCGetNumPortURLs ( PORT_HANDLE hPort,
                           uint32_t * piNumURLs )
```

Inquires the number of available URLs for this port.

Parameters

- [in] *hPort* Module or remote device port handle to access Port from.
- [out] *piNumURLs* Number of available URL entries.

Returns

- GC_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).

GC_ERR_INVALID_HANDLE	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
GC_ERR_INVALID_PARAMETER	Parameter <i>piNumURLs</i> is an invalid pointer (NULL or ~0x0).
GC_ERR_NOT_IMPLEMENTED	The provided module handle does not have a Port module implemented.
GC_ERR_IO	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.4 GCGetPortURLInfo

```

GC ERROR  GCGetPortURLInfo    ( PORT_HANDLE    hPort,
                               uint32_t            iURLIndex,
                               URL_INFO_CMD        iInfoCmd,
                               INFO_DATATYPE *    piType,
                               void *             pBuffer,
                               size_t *           piSize )
    
```

Queries detailed port information as defined in [URL_INFO_CMD](#).

In case a module does not support multiple URLs and/or the related information the function will return GC_ERR_NOT_AVAILABLE for information which cannot be retrieved.

Parameters

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
GC_ERR_INVALID_INDEX	<i>iURLIndex</i> is greater than the number available URLs -1.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented or the provided module handle does not have a Port module implemented.
GC_ERR_NOT_AVAILABLE	The module does not provide the requested information.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0).
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_IO	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.5 GCReadPort

```

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 endianness 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. For example if the underlying technology only allows a 4-byte 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.

The function is used to handle GenICam GenApi port read access when it is in general unknown which type of data is being read and whether it is acceptable to read only part of the requested *piSize* bytes. The operation is therefore considered successful only if all requested

data has been read. The GenTL Producer is not allowed to report success (GC_ERR_SUCCESS) if the operation was finished only partially.

Parameters

[in] <i>hPort</i>	Module or remote device port handle to access Port from.
[in] <i>iAddress</i>	Byte address to read from.
[out] <i>pBuffer</i>	Pointer to a user allocated byte buffer to receive data; this must not be NULL.
[in,out] <i>piSize</i>	Size of the provided <i>pBuffer</i> 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 (if that is different from the requested size, the function return value should indicate the reason).

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
GC_ERR_INVALID_PARAMETER	Parameters <i>pBuffer</i> and/or <i>piSize</i> are invalid pointers (NULL or ~0x0).
GC_ERR_ACCESS_DENIED	The access to the requested register <i>iAddress</i> is denied because the register is not readable.
GC_ERR_INVALID_ADDRESS	<i>iAddress</i> is invalid for example because the port's register space is only 32Bit wide and <i>iAddress</i> is in the 64Bit register space or because there is no register with the provided <i>iAddress</i> .
GC_ERR_NOT_IMPLEMENTED	The provided module handle does not have a Port module implemented.
GC_ERR_IO	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.6 GCWritePort

GC_ERROR	GCWritePort	(PORT_HANDLE uint64_t const void * size_t *	<i>hPort</i> , <i>iAddress</i> , <i>pBuffer</i> , <i>piSize</i>)
--------------------------	-------------	---	--

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 endianness 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. 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).

The function is used to handle GenICam GenApi port write access when it is in general unknown which type of data is being written and whether it is acceptable to write only part of the requested *piSize* bytes. The operation is therefore considered successful only if all requested data has been written. The GenTL Producer is not allowed to report success (GC_ERR_SUCCESS) if the operation was finished only partially.

Parameters

[in] <i>hPort</i>	Module or remote device port handle to access the Port from.
[in] <i>iAddress</i>	Byte address to write to.
[in] <i>pBuffer</i>	Pointer to a user allocated byte buffer containing the data to write; this must not be NULL.
[in,out] <i>piSize</i>	Size of the provided <i>pBuffer</i> 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 (if that is different from the requested size, the function return value should indicate the reason).

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
GC_ERR_INVALID_PARAMETER	Parameters <i>pBuffer</i> and/or <i>piSize</i> are invalid pointers (NULL or ~0x0).
GC_ERR_ACCESS_DENIED	The access to the requested register <i>iAddress</i> is denied because the register is not writable or because the Port

	module is opened in a way that it does not allow write access.
GC_ERR_INVALID_ADDRESS	<i>iAddress</i> is invalid for example because the port's register space is only 32Bit wide and <i>iAddress</i> is in the 64Bit register space or because there is no register with the provided <i>iAddress</i> .
GC_ERR_NOT_IMPLEMENTED	The provided module handle does not have a Port module implemented.
GC_ERR_INVALID_VALUE	An invalid value has been written. This error code is to be returned if the underlying registermap provides that information. In case the underlying technology does not provide that level of information a GC_ERR_ACCESS_DENIED is to be returned.
GC_ERR_IO	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.7 GCWritePortStacked

```
GC ERROR GCWritePortStacked ( PORT_HANDLE hPort,
                             PORT_REGISTER_STACK_ENTRY *
                             pEntries,
                             size_t * piNumEntries )
```

Writes a number of bytes to the given address on the specified *hPort* for every element in the *pEntries* array. The endianness 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. 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).

In case of an error the function returns the number of successful writes in *piNumEntries* even though it returns an error code as return value. This is an exception to the statement in the section Error Handling (see chapter [6.1.5](#)).

Parameters

[in] <i>hPort</i>	Module or remote device port handle to access the Port from.
[in] <i>pEntries</i>	Array of structures containing write address and data to write.
[in,out] <i>piNumEntries</i>	In: Number of entries in the array, Out: Number of successful executed writes according to the entries in the <i>pEntries</i> array.

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .
<code>GC_ERR_INVALID_HANDLE</code>	The handle <i>hPort</i> is invalid (NULL) or does not reference an open module.
<code>GC_ERR_INVALID_PARAMETER</code>	Parameters <i>pEntries</i> and/or <i>piNumEntries</i> are invalid pointers (NULL or ~0x0).
<code>GC_ERR_ACCESS_DENIED</code>	The access to at least one of the requested registers is denied because the register is not writable or because the Port module is opened in a way that it does not allow write access.
<code>GC_ERR_NOT_IMPLEMENTED</code>	The provided module handle does not have a Port module implemented.
<code>GC_ERR_INVALID_ADDRESS</code>	One or more entries in <i>pEntries</i> has an invalid address for example because the port's register space is only 32Bit wide and <i>Address</i> is in the 64Bit register space or because there is no register with the specified address.
<code>GC_ERR_INVALID_VALUE</code>	An invalid value has been written. This error code is to be returned if the underlying registermap provides that information. In case the underlying technology/registermap does not provide that level of information a <code>GC_ERR_ACCESS_DENIED</code> is to be returned.
<code>GC_ERR_IO</code>	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.6.8 GCReadPortStacked

```
GC_ERROR GCReadPortStacked ( PORT_HANDLE    hPort,
                             PORT_REGISTER_STACK_ENTRY *
                             pEntries,
                             size_t * piNumEntries )
```

Reads a number of bytes from the given address on the specified *hPort* for every element in the *pEntries* array. The endianness of the data content is specified by the [GCGetPortInfo](#) function.

If the underlying technology has alignment restrictions on the port access the GenTL Provider implementation has to handle this internally. For example if the underlying technology only

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

allows a `uint32_t` aligned access and the calling GenTL Consumer wants to read 5 bytes starting at address 2. The implementation has to read 8 bytes starting at address 0 and to extract the 5 bytes requested.

In case of an error the function returns the number of successful reads in `piNumEntries` even though it returns an error code as return value. This is an exception to the statement in the section Error Handling.

Parameters

[in] <code>hPort</code>	Module or remote device port handle to access the Port from.
[in] <code>pEntries</code>	Array of structures containing read address and data to read.
[in,out] <code>piNumEntries</code>	In: Number of entries in the array, Out: Number of successful executed reads according to the entries in the <code>pEntries</code> array.

Returns

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_INITIALIZED</code>	No preceding call to GCInitLib .
<code>GC_ERR_INVALID_HANDLE</code>	The handle <code>hPort</code> is invalid (NULL) or does not reference an open module.
<code>GC_ERR_INVALID_PARAMETER</code>	Parameters <code>pEntries</code> and/or <code>piNumEntries</code> are invalid pointers (NULL or ~0x0).
<code>GC_ERR_ACCESS_DENIED</code>	The access to at least one of the requested registers is denied because the register is not readable or because the Port module is opened in a way that it does not allow read access.
<code>GC_ERR_NOT_IMPLEMENTED</code>	The provided module handle does not have a Port module implemented.
<code>GC_ERR_INVALID_ADDRESS</code>	One or more addresses in the entries in <code>pEntries</code> has an invalid address for example because the port's register space is only 32Bit wide and <code>Address</code> is in the 64Bit register space or because there is no register with the specified address.
<code>GC_ERR_IO</code>	Communication error or connection lost.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7 Signaling Functions

6.3.7.1 EventFlush

```
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_ERR_SUCCESS Operation was successful; no error occurred.
 GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
 GC_ERR_INVALID_HANDLE The handle *hEvent* is invalid (NULL) or does not reference a previously registered event.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.2 EventGetData

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

Retrieves the next event data entry 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.

In case of a New Buffer event the `EventGetData` function return the [EVENT_NEW_BUFFER_DATA](#) structure in the provided buffer.

In case `EventGetData` returns an error (for example `GC_ERR_ABORT`) no event is removed from the internal queue and the event stays signaled. Event counters are not affected.

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 case of a New Buffer Event being retrieved with *pBuffer* equals NULL the buffer is removed from the output queue and not requested.

[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. If set to [GENTL_INFINITE](#) the timeout is infinite and the function will only return if the operation is completed or if [EventKill](#) is called on this event object.
A value of 0 checks the state of the event object and returns immediately either with a timeout or with event data.

Returns

- GC_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE The handle *hEvent* is invalid (NULL) or does not reference a previously registered event.
- GC_ERR_INVALID_PARAMETER Parameter *piSize* is an invalid pointer (NULL or ~0x0).
- GC_ERR_BUFFER_TOO_SMALL *pBuffer* is not NULL and the value of **piSize* is too small to receive the expected amount of data.
- GC_ERR_ABORT The current wait operation has been terminated through a call to [EventKill](#).
- GC_ERR_TIMEOUT The specified *iTimeout* expired before the event *hEvent* occurred.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.3 EventGetDataInfo

```

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]	<i>hEvent</i>	Event handle to parse data from.
[in]	<i>pInBuffer</i>	Pointer to a buffer containing event data. This value must not be NULL.
[in]	<i>iInSize</i>	Size of the provided <i>pInBuffer</i> in bytes.
[in]	<i>iInfoCmd</i>	Information to be retrieved as defined in EVENT DATA INFO CMD and EVENT TYPE .
[out]	<i>piType</i>	Data type of the <i>pOutBuffer</i> content as defined in the EVENT DATA INFO CMD , EVENT TYPE and INFO DATATYPE .
[in,out]	<i>pOutBuffer</i>	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>piOutSize</i> will contain the minimal size of <i>pOutBuffer</i> in bytes. If the <i>piType</i> is a string the size includes the terminating 0.
[in,out]	<i>piOutSize</i>	<p><i>pOutBuffer</i> equal NULL: out: minimal size of <i>pOutBuffer</i> in bytes to hold all information.</p> <p><i>pOutBuffer</i> unequal NULL: in: size of the provided <i>pOutBuffer</i> in bytes. out: number of bytes filled by the function.</p>

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hEvent</i> is invalid (NULL or ~0x0) or does not reference a previously registered event.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>pInBuffer</i> , <i>piOutSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0) or <i>iInSize</i> is 0
GC_ERR_BUFFER_TOO_SMALL	<i>pOutBuffer</i> is not NULL and the value of <i>*piOutSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.4 EventGetInfo

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

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hEvent</i> is invalid (NULL) or does not reference a previously registered event.
GC_ERR_NOT_IMPLEMENTED	Specified <i>iInfoCmd</i> is not implemented.
GC_ERR_INVALID_PARAMETER	Parameters <i>piSize</i> and/or <i>piType</i> are invalid pointers (NULL or ~0x0)
GC_ERR_BUFFER_TOO_SMALL	<i>pBuffer</i> is not NULL and the value of <i>*piSize</i> is too small to receive the expected amount of data.
GC_ERR_NOT_AVAILABLE	The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.5 EventKill

GC_ERROR EventKill	(EVENT_HANDLE <i>hEvent</i>)
------------------------------------	--------------------------------

Terminates a waiting operation on a previously registered event object. In case of multiple pending wait operations EventKill causes one wait operation to return with a [GC_ERR_ABORT](#) error code. Therefore in order to cancel all pending wait operations EventKill must be called as many times as wait operations are pending. In case this function is called while no wait operation was pending the next call to [EventGetData](#) will return a [GC_ERR_ABORT](#). This behavior can be cleared by unregistering and reregistering the event.

In case there are pending events in the queue the EventKill has higher priority and on the pending/next call to [EventGetData](#) a [GC_ERR_ABORT](#) is returned.

EventKill does not free any resources.

Parameters

[in] *hEvent* Handle to event object.

Returns

- GC_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_INITIALIZED No preceding call to [GCInitLib](#).
- GC_ERR_INVALID_HANDLE The handle *hEvent* is invalid (NULL) or does not reference a previously registered event.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.6 GCRegisterEvent

GC_ERROR GCRegisterEvent	(EVENTSRC_HANDLE <i>hModule</i> , EVENT_TYPE <i>iEventID</i> , EVENT_HANDLE * <i>phEvent</i>)
--	---

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. It is recommended to initialize **phEvent* to

[GENTL_INVALID_HANDLE](#) before calling [GCRegisterEvent](#) to indicate an invalid handle.

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
GC_ERR_NOT_INITIALIZED	No preceding call to GCInitLib .
GC_ERR_INVALID_HANDLE	The handle <i>hModule</i> is invalid (NULL) or does not reference a previously instantiated module.
GC_ERR_RESOURCE_IN_USE	The given <i>iEventID</i> has been registered before on the given <i>hModule</i> .
GC_ERR_NOT_IMPLEMENTED	The specified event type is not implemented in the provided module of the GenTL Producer. Applies also if specified <i>iEventID</i> is not a valid event type for given module.
GC_ERR_NOT_AVAILABLE	The specified event type is not available in the provided module <i>hModule</i> (for example because the remote device does not implement it).
GC_ERR_INVALID_PARAMETER	Parameter <i>phEvent</i> is an invalid pointer (NULL or ~0x0)

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.3.7.7 GCUnregisterEvent

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

A call to this function will unregister the given *iEventID* from the given *hModule*. This will terminate all pending wait operations of [EventGetData](#) with the error code [GC_ERR_ABORT](#). Pending events are silently discarded.

For the [EVENT_NEW_BUFFER](#) all pending buffers in the output queue are set to a non queued state to match the behavior of normal events. All buffers in the input pool or buffers currently being filled are not touched.

Parameters

[in] <i>hModule</i>	Module handle to unregister event with.
[in] <i>iEventID</i>	Event type to unregister as defined in EVENT_TYPE .

Returns

GC_ERR_SUCCESS	Operation was successful; no error occurred.
----------------	--

GC_ERR_NOT_INITIALIZED	The event has not previously been registered through GCRegisterEvent or no preceding call to GCInitLib has been made.
GC_ERR_INVALID_HANDLE	The handle <i>hModule</i> is invalid (NULL) or does not reference a previously instantiated module.
GC_ERR_NOT_IMPLEMENTED	The specified event type is not implemented in the provided module of the GenTL Producer.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.4 Enumerations

Enumeration values are signed 32 bit integers.

6.4.1 Library and System Enumerations

6.4.1.1 INFO_DATATYPE

```
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 (encoding according to the <code>TL_INFO_CHAR_ENCODING</code> info command).
INFO_DATATYPE_STRINGLIST	2	Concatenated <code>INFO_DATATYPE_STRING</code> 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 (<i>void*</i>). Size is platform dependent (32 bit on 32 bit platforms)
INFO_DATATYPE_BOOL8	11	Boolean value occupying 8 bit. 0 for <i>false</i>

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Enumerator	Value	Description
		and anything for <code>true</code> .
<code>INFO_DATATYPE_SIZET</code>	12	Platform dependent unsigned integer (32 bit on 32 bit platforms)
<code>INFO_DATATYPE_BUFFER</code>	13	Like a <code>INFO_DATATYPE_STRING</code> but with arbitrary data and no 0 termination.
<code>INFO_DATATYPE_PTRDIFF</code>	14	The type <code>ptrdiff_t</code> is a type that can hold the result of subtracting two pointers.
<code>INFO_DATATYPE_CUSTOM_ID</code>	1000	Starting value for Custom IDs which are implementation specific. If a generic GenTL Consumer is using custom data types provided through a specific GenTL Producer implementation it must differentiate the handling of GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.1.2 TL_CHAR_ENCODING_LIST

```
enum TL_CHAR_ENCODING_LIST
```

Char encoding schemata.

Enumerator	Value	Description
<code>TL_CHAR_ENCODING_ASCII</code>	0	Char encoding of the GenTL Producer is ASCII.
<code>TL_CHAR_ENCODING_UTF8</code>	1	Char encoding of the GenTL Producer is UTF8.

6.4.1.3 TL_INFO_CMD

```
enum TL_INFO_CMD
```

System module information commands for the [TLGetInfo](#) and [GCGetInfo](#) functions. The reported information through these two functions must be in sync.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M) or optional (O).

Enumerator	Impl	Value	Description
<code>TL_INFO_ID</code>	M	0	Unique ID identifying a GenTL Producer. For example the filename of the GenTL Producer implementation including its path.

Enumerator	Impl	Value	Description
			Data type: STRING
TL_INFO_VENDOR	M	1	GenTL Producer vendor name. Data type: STRING
TL_INFO_MODEL	M	2	GenTL Producer model name. For example: A vendor produces more than one GenTL Producer for different device classes or different technologies. The TL_INFO_MODEL references a single GenTL Producer implementation. The combination of Vendor and Model provides a unique reference of ONE GenTL Producer implementation. Data type: STRING
TL_INFO_VERSION	M	3	GenTL Producer version. Data type: STRING
TL_INFO_TLTYPE	M	4	Transport layer technology that is supported. See string constants in chapter 6.6.1. Data type: STRING
TL_INFO_NAME	M	5	File name including extension of the library. Data type: STRING
TL_INFO_PATHNAME	M	6	Full path including file name and extension of the library. Data type: STRING
TL_INFO_DISPLAYNAME	M	7	User readable name of the GenTL Producer. Data type: STRING
TL_INFO_CHAR_ENCODING	M	8	The char encoding of the GenTL Producer. Data type: INT32 (TL_CHAR_ENCODING_LIST enumeration value) Data type: INT32
TL_INFO_GENTL_VER_MAJOR	M	9	Major version number of GenTL Standard Version this Producer complies with. Data type: UINT32
TL_INFO_GENTL_VER_MINOR	M	10	Minor version number of GenTL Standard Version this Producer complies with. Data type: UINT32
TL_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer

Enumerator	Impl	Value	Description
			<p>custom IDs which are implementation specific.</p> <p>If a generic GenTL Consumer is using custom TL_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.</p>

6.4.2 Interface Enumerations

6.4.2.1 INTERFACE_INFO_CMD

```
enum INTERFACE_INFO_CMD
```

This enumeration defines commands to retrieve information with the [IFGetInfo](#) function from the Interface module or through [TLGetInterfaceInfo](#).

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M) or optional (O).

Enumerator	Impl	Value	Description
INTERFACE_INFO_ID	M	0	Unique ID of the interface. Data type: STRING
INTERFACE_INFO_DISPLAYNAME	M	1	User readable name of the interface. Data type: STRING
INTERFACE_INFO_TLTYPE	M	2	Transport layer technology that is supported. See string constants in chapter 6.6.1. Data type: STRING
INTERFACE_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom INTERFACE_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.3 Device Enumerations

6.4.3.1 DEVICE_ACCESS_FLAGS

```
enum DEVICE_ACCESS_FLAGS
```

This enumeration defines different modes how a device is to be opened with the [IFOpenDevice](#) function. The values can not be combined.

Enumerator	Value	Description
DEVICE_ACCESS_UNKNOWN	0	Not used in a command. It can be used to initialize a variable to query that information.
DEVICE_ACCESS_NONE	1	Deprecated: do not use. <i>(This value was never usable in any meaningful way.)</i>
DEVICE_ACCESS_READONLY	2	Opens the device read only. All Port functions can only read from the device.
DEVICE_ACCESS_CONTROL	3	Opens 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 which are implementation specific. If a generic GenTL Consumer is using custom DEVICE_ACCESS_FLAGSs provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.3.2 DEVICE_ACCESS_STATUS

```
enum DEVICE_ACCESS_STATUS
```

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

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 to be opened for Read/Write access but it is currently not opened. This value will only be returned through IFGetDeviceInfo function because as soon as the device is open <code>DEVICE_ACCESS_STATUS_OPEN_READWRITE</code> will be returned.
DEVICE_ACCESS_STATUS_READONLY	2	The device is available to be opened for Read access but is currently not opened. In case the device allows both read and write access the value <code>DEVICE_ACCESS_STATUS_READWRITE</code> has a higher priority. This value will only be returned through IFGetDeviceInfo function because as soon as the device is open <code>DEVICE_ACCESS_STATUS_OPEN_READONLY</code> will be returned.
DEVICE_ACCESS_STATUS_NOACCESS	3	The device is seen by the producer but is not available for access because it is not reachable.
DEVICE_ACCESS_STATUS_BUSY	4	The device is already owned/opened by another entity.
DEVICE_ACCESS_STATUS_OPEN_READWRITE	5	The device is already owned/opened by this GenTL Producer with RW access. A further call to IFOpenDevice will return <code>GC_ERR_RESOURCE_IN_USE</code> .
DEVICE_ACCESS_STATUS_OPEN_READONLY	6	The device is already owned/opened by this GenTL Producer with RO access. A further call to IFOpenDevice will return <code>GC_ERR_RESOURCE_IN_USE</code> .
DEVICE_ACCESS_STATUS_CUSTOM_ID	1000	Starting value for custom IDs which are implementation specific. If a generic GenTL Consumer is using custom <code>DEVICE_ACCESS_STATUS</code> ids provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.3.3 DEVICE_INFO_CMD

enum DEVICE_INFO_CMD

This enumeration defines commands to retrieve information with the [DevGetInfo](#) function on a device handle or with [IFGetDeviceInfo](#). The reported information using these two functions should be in sync if the information is available. This is also true for the info command [DEVICE_INFO_ACCESS_STATUS](#).

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
DEVICE_INFO_ID	M	0	Unique ID of the device. Data type: STRING
DEVICE_INFO_VENDOR	M	1	Device vendor name. Data type: STRING
DEVICE_INFO_MODEL	M	2	Device model name. Data type: STRING
DEVICE_INFO_TLTYPE	M	3	Transport layer technology that is supported. See string constants in chapter 6.6.1 . Data type: STRING
DEVICE_INFO_DISPLAYNAME	M	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	O	5	Gets the access status the GenTL Producer has on the device. Data type: INT32 (DEVICE_ACCESS_STATUS enumeration value)
DEVICE_INFO_USER_DEFINED_NAME	O	6	String containing the user defined name of the device. If the information is not available, the query should result in GC_ERR_NOT_AVAILABLE. Data type: STRING
DEVICE_INFO_SERIAL_NUMBER	CM	7	Serial number of the device in string format. If the information is not available, the query should result in GC_ERR_NOT_AVAILABLE. Data type: STRING

Enumerator	Impl	Value	Description
DEVICE_INFO_VERSION	O	8	Device version in string format. If the information is not known, the query should result in GC_ERR_NOT_AVAILABLE. Data type: STRING
DEVICE_INFO_TIMESTAMP_FREQUENCY	O	9	Tick frequency of the device's timestamp counter in ticks per second. The counter is used for example to assign timestamps to the individual acquired buffers (BUFFER_INFO_TIMESTAMP). Data type: UINT64
DEVICE_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom DEVICE_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.4 Data Stream Enumerations

6.4.4.1 ACQ_QUEUE_TYPE

```
enum ACQ_QUEUE_TYPE
```

This enumeration commands define from which to which queue/pool buffers are flushed with the [DSFlushQueue](#) function.

Enumerator	Value	Description
------------	-------	-------------

Enumerator	Value	Description
ACQ_QUEUE_INPUT_TO_OUTPUT	0	Flushes the buffers from the input pool to the output buffer queue and if necessary adds entries in the “New Buffer” event data queue. The buffers currently being filled are not affected by this operation. This only applies to the mandatory default buffer handling mode. Whether the buffer contains real data or is result of a flush operation can be inquired through the buffer info command BUFFER_INFO_NEW_DATA . This allows the GenTL Consumer to maintain all buffers without a second reference in the GenTL Consumer because all buffers are delivered through the new buffer event.
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. This is including those in the output buffer queue and the ones which are currently being filled and discard entries in the event data queue.
ACQ_QUEUE_UNQUEUED_TO_INPUT	3	Puts all buffers that are neither in the input pool nor being currently filled nor in the output buffer queue in the input pool.
ACQ_QUEUE_ALL_DISCARD	4	Discards all buffers in the input pool and the buffers in the output queue including buffers currently being filled so that no buffer is bound to any internal mechanism and all buffers may be revoked or requeued.
ACQ_QUEUE_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom ACQ_QUEUE_TYPES provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

6.4.4.2 ACQ_START_FLAGS

```
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.

6.4.4.3 ACQ_STOP_FLAGS

```
enum ACQ_STOP_FLAGS
```

This enumeration defines special stop flags for the acquisition engine. The function used is [DSStopAcquisition](#).

Enumerator	Value	Description
ACQ_STOP_FLAGS_DEFAULT	0	Stops 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. In case this results in a partially filled buffer the Producer will return the buffer through the regular mechanism to the user, indicating through the info function of that buffer that this buffer is not complete.
ACQ_STOP_FLAGS_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom ACQ_STOP_FLAGS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.4.4 BUFFER_INFO_CMD

```
enum BUFFER_INFO_CMD
```

This enumeration defines commands to retrieve information with the [DSGetBufferInfo](#) function on a buffer handle. In case a BUFFER_INFO_CMD is not available or not

implemented the [DSGetBufferInfo](#) function must return the appropriate error return value.

For multi-part buffers it is possible to query information on each part. Therefore some of the BUFFER_INFO_COMMANDS are not used or overwritten by [BUFFER PART INFO COMMANDS](#). The enumeration table below lists which commands are applicable on the global buffer if the underlying buffer contains multi-part data. In the table listing the command values the column “Global-Part/Impl” lists if a given info command is to be queried on the global buffer or if the command is over written by an info command of the image part within the buffer. The possible values are:

Acronym	Description
G/O	The information is to be inquired on the global buffer. Implementation of the command is recommended but optional/technology dependent.
G/M	The information is to be inquired on the global buffer. Implementation of the command is mandatory. In case a similar command is available for buffer part also the scope of the BUFFER_INFO_COMMAND is the global buffer where the BUFFER PART INFO COMMAND is describing the part.
G/CM	The information is to be inquired on the global buffer. Implementation of the command is conditional mandatory. Conditional mandatory is used for commands which might not always be applicable. If it is possible to implement a certain command it must be implemented. In case a similar command is available for buffer part also the scope of the BUFFER_INFO_COMMAND is the global buffer where the BUFFER PART INFO COMMAND is describing the part.
P/O	The command is not available in case the buffer contains multi-part data. In this case the function DSGetBufferInfo returns GC_ERR_NO_DATA. In case the buffer does not contain multi-part data the command returns the requested information. The implementation of the command is optional.
P/M	The command is not available in case the buffer contains multi-part data. In this case the function DSGetBufferInfo returns GC_ERR_NO_DATA. In case the buffer does not contain multi-part data the command returns the requested information. In this case the implementation of the command is mandatory .

Acronym	Description
P/CM	<p>The command is not available in case the buffer contains multi-part data. In this case the function DSGetBufferInfo returns GC_ERR_NO_DATA.</p> <p>In case the buffer does not contain multi-part data the command returns the requested information. In this case the implementation of the command is conditional mandatory. Conditional mandatory is used for commands which might not always be applicable. If it is possible to implement a certain command it must be implemented.</p>

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_BASE	G/M	0	<p>Base address of the buffer memory as passed to the DSAnnounceBuffer function or allocated by DSAllocAndAnnounceBuffer. This is also the address where the payload within the buffer starts. This info is irrelevant for composite buffers announced using DSAnnounceCompositeBuffer, function returns GC_ERR_NO_DATA. Data type: PTR</p>
BUFFER_INFO_SIZE	G/M	1	<p>Size of the buffer in bytes. For composite buffers announced using DSAnnounceCompositeBuffer, this returns the sum of sizes of the composite buffer segments. Data type: SIZET</p>
BUFFER_INFO_USER_PTR	G/O	2	<p>Private data pointer casted to an integer provided at buffer announcement using one of the buffer announcement functions by the GenTL Consumer. The pointer is attached to the buffer to allow attachment of user data to a buffer. Data type: PTR</p>

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_TIMESTAMP	G/O	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	G/M	4	Flag to indicate that the buffer contains new data since the last delivery. Value 0 means this buffer has not been processed by the acquisition engine (it got delivered e.g. as a result of a flush operation). In such case other data related buffer info queries may fail. However, data agnostic information (such as BUFFER_INFO_BASE, BUFFER_INFO_SIZE, or BUFFER_INFO_USER_PTR) must still be correctly reported by GenTL Producer. Data type: BOOL8
BUFFER_INFO_IS_QUEUED	G/M	5	If this flag is set to true the buffer is in the input pool, the buffer is currently being filled or the buffer is in the output buffer queue. In case this value is true the buffer is owned by the GenTL Producer and it can not be revoked. Data type: BOOL8
BUFFER_INFO_IS_ACQUIRING	G/CM	6	Flag to indicate that the buffer is currently being filled with data. Data type: BOOL8
BUFFER_INFO_IS_INCOMPLETE	G/M	7	Flag to indicate that a buffer was filled but an error occurred during that process. Data type: BOOL8
BUFFER_INFO_TLTYPE	G/M	8	Transport layer technology that is supported. See string constants in chapter 6.6.1. Data type: STRING

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_SIZE_FILLED	G/O	9	<p>Number of bytes written into the buffer the last time it has been filled. This value is reset to 0 when the buffer is placed into the Input Buffer Pool. If the buffer is incomplete (such as if there are missing packets), only the number of bytes successfully written to the buffer are reported. If the buffer is complete and payload contains no gaps, the number equals to the size reported through BUFFER_INFO_DATA_SIZE. Note that gaps in payload are possible with certain payload types, such as GenDC or multi-part.</p> <p>Data type: SIZET</p>
BUFFER_INFO_WIDTH	P/CM	10	<p>Width of the data in the buffer in number of pixels. This information refers for example to the width entry in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly.</p> <p>Data type: SIZET</p>
BUFFER_INFO_HEIGHT	P/CM	11	<p>Height of the data in the buffer in number of pixels as configured. For variable size images this is the maximum height of the buffer. For example this information refers to the height entry in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly.</p> <p>Data type: SIZET</p>
BUFFER_INFO_XOFFSET	P/CM	12	<p>XOffset of the data in the buffer in number of pixels from the image origin to handle areas of interest. This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly.</p> <p>Data type: SIZET</p>

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_YOFFSET	P/CM	13	YOffset of the data in the buffer in number of lines from the image origin to handle areas of interest. This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly. Data type: SIZET
BUFFER_INFO_XPADDING	P/CM	14	XPadding of the data in the buffer in number of bytes. This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is may be implemented accordingly. Data type: SIZET
BUFFER_INFO_YPADDING	G/O	15	YPadding of the data in the buffer in number of bytes. This information refers for example to the information provided in the GigE Vision image stream data leader. For other thechnologies this may be implemented accordingly. Data type: SIZET
BUFFER_INFO_FRAMEID	G/M	16	A sequentially incremented number of the frame. This information refers for example to the information provided in the GigE Vision image stream block id. For other technologies this is to be implemented accordingly. The wrap around of this number is transportation technology dependent. For GigE Vision it is (so far) 16bit wrapping to 1. Other technologies may implement a larger bit depth. Data type: UINT64

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_IMAGEPRESENT	G/M	17	Flag to indicate if the current data in the buffer contains single raster scan image data. This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly. The flag should report true if the payload contains a single raster scan image in the sense of traditional image payload type and if it makes sense to query its properties using buffer-global image format related buffer infos. Data type: BOOL8
BUFFER_INFO_IMAGEOFFSET	G/O	18	Offset of the image data from the beginning of the delivered buffer in bytes. Applies for example when delivering the image as part of chunk data or on technologies requiring specific buffer alignment. Data type: SIZET
BUFFER_INFO_PAYLOADTYPE	G/M	19	Payload type of the data. This information refers to the constants defined in PAYLOADTYPE INFO IDS . Data type: SIZET
BUFFER_INFO_PIXELFORMAT	P/CM	20	Pixelformat of the data. This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly. The interpretation of the pixel format depends on the namespace the pixel format belongs to. This can be inquired using the <code>BUFFER_INFO_PIXELFORMAT_NAMESPACE</code> command. Data type: UINT64

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_PIXELFORMAT_NAMESPACE	P/CM	21	This information refers to the constants defined in PIXELFORMAT_NAMESPACE_IDS to allow interpretation of BUFFER_INFO_PIXELFORMAT. Data type: UINT64
BUFFER_INFO_DELIVERED_IMAGEHEIGHT	P/CM	22	The number of lines in the current buffer as delivered by the transport mechanism. For area scan type images this is usually the number of lines configured in the device. For variable size linescan images this number may be lower than the configured image height. This information refers for example to the information provided in the GigE Vision image stream data trailer. For other technologies this is to be implemented accordingly. Data type: SIZET
BUFFER_INFO_DELIVERED_CHUNKPAYLOADSIZE	G/CM	23	This information refers for example to the information provided in the GigE Vision image stream data trailer. For other technologies this is to be implemented accordingly. For GenDC payload the eventual chunk data must be processed based on the information in the GenDC descriptor, the GenTL Producer must return GC_ERR_NO_DATA in that case. Data type: SIZET

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_CHUNKLAYOUTID	G/CM	24	<p>This information refers for example to the information provided in the GigE Vision image stream data leader. The chunk layout id serves as an indicator that the chunk layout has changed and the application should re-parse the chunk layout in the buffer. When a chunk layout (availability or position of individual chunks) changes since the last buffer delivered by the device through the same stream, the device must change the chunk layout id. As long as the chunk layout remains stable, the camera must keep the chunk layout id intact. When switching back to a layout, which was already used before, the camera can use the same id again or use a new id. A chunk layout id value of 0 is invalid. It is reserved for use by cameras not supporting the layout id functionality. The algorithm used to compute the chunk layout id is left as quality of implementation. For other technologies this is to be implemented accordingly. For GenDC payload the eventual chunk data must be processed based on the information in the GenDC descriptor, the GenTL Producer must return GC_ERR_NO_DATA in that case. Data type: UINT64</p>

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_FILENAME	G/CM	25	<p>Filename in case the payload contains a file.</p> <p>This information refers for example to the information provided in the GigE Vision image stream data leader. For other technologies this is to be implemented accordingly. Since this is GigE Vision related information and the filename in GigE Vision is UTF8 coded, this filename is also UTF8 coded.</p> <p>Data type: STRING</p>
BUFFER_INFO_PIXEL_ENDIANNESS	G/O	26	<p>Endianness of the multi-byte pixel data in the buffer. This information refers to the constants defined in PIXELENDIANNESS_IDS.</p> <p>Data type: INT32</p>
BUFFER_INFO_DATA_SIZE	G/O	27	<p>Size of the data intended to be written to the buffer last time it has been filled. This value is reset to 0 when the buffer is placed into the Input Buffer Pool.</p> <p>If the buffer is incomplete the number still reports the full size of the original data including the lost parts. If the buffer is complete and the payload contains no gaps, the number equals to the size reported through <code>BUFFER_INFO_SIZE_FILLED</code>.</p> <p>Note that gaps in payload are possible with certain payload types, such as GenDC or multi-part.</p> <p>Data type: SIZET</p>
BUFFER_INFO_TIMESTAMP_NS	G/O	28	<p>Timestamp the buffer was acquired, in units of 1 ns (1 000 000 000 ticks per second). If the device is internally using another tick frequency than 1GHz, the GenTL Producer must convert the value to nanoseconds.</p> <p>Data type: UINT64</p>

Enumerator	Global -Part /Impl	Value	Description
BUFFER_INFO_DATA_LARGER_THAN_BUFFER	G/O	29	If this values is set to true it indicates that the payload transferred would not fit into the announced buffer and that therefore only parts of the payload or no payload (depending on the implementation of the GenTL Producer) is copied into the buffer. Data type: BOOL8
BUFFER_INFO_CONTAINS_CHUNKDATA	G/M	30	If this values is set to true it indicates that the payload transferred contains chunk data which may be parsed through a call to DSGetBufferChunkData or the GenTL Consumer. For GenDC payload the eventual chunk data must be processed based on the information in the GenDC descriptor, the GenTL Producer must return GC_ERR_NO_DATA in that case. Data type: BOOL8
BUFFER_INFO_IS_COMPOSITE	G/M	31	Indicates whether this is a composite buffer, announced using DSAnnounceCompositeBuffer . Data type: BOOL8
BUFFER_INFO_CUSTOM_ID	G/O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom BUFFER_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.4.5 PAYLOADTYPE_INFO_IDS

enum PAYLOADTYPE_INFO_IDS

This enumeration defines constants to give a hint on the payload type to be expected in the buffer. These values are returned by a call to [DSGetBufferInfo](#) with the command [BUFFER_INFO_PAYLOADTYPE](#). The interpretation of the PAYLOADTYPE_INFO_IDS is depending on the TLType of the device which streams the data.

Enumerator	Value	Description
PAYLOAD_TYPE_UNKNOWN	0	The GenTL Producer is not aware of the payload type of the data in the provided buffer. For the GenTL Consumer perspective this can be handled as raw data.
PAYLOAD_TYPE_IMAGE	1	The buffer payload contains image data. The GenTL Consumer can check if additional chunk data is available via the BUFFER_INFO commands.
PAYLOAD_TYPE_RAW_DATA	2	The buffer payload contains raw and further unspecified data. This can be used to send acquisition statistics.
PAYLOAD_TYPE_FILE	3	The buffer payload contains data of a file. It is used to transfer files such as JPEG compressed images which can be stored by the GenTL Producer directly to a hard disk. The user might get a hint how to interpret the buffer by the filename provided through a call to DSGetBufferInfo with the command BUFFER_INFO_FILENAME .
PAYLOAD_TYPE_CHUNK_DATA	4	The buffer payload contains chunk data which can be parsed. The chunk data type might be reported through SFNC or deduced from the technology the device is based on. This constant is for backward compatibility with GEV 1.2 and is deprecated since GenTL version 1.5. From now on ChunkData can be part or any other payload type. Use the BUFFER_INFO_CONTAINS_CHUNKDATA commands to query if a given buffer content contains chunk data.
PAYLOAD_TYPE_JPEG	5	The buffer payload contains JPEG data in the format described in GEV 2.0. The GenTL Producer should report additional information through the corresponding BUFFER_INFO_CMD commands.


Enumerator	Value	Description
PAYLOAD_TYPE_JPEG2000	6	The buffer payload contains JPEG 2000 data in the format described in GEV 2.0. The GenTL Producer should report additional information through the corresponding BUFFER_INFO_CMD commands.
PAYLOAD_TYPE_H264	7	The buffer payload contains H.264 data in the format described in GEV 2.0. The GenTL Producer should report additional information through the corresponding BUFFER_INFO_CMD commands.
PAYLOAD_TYPE_CHUNK_ONLY	8	The buffer payload contains only chunk data but no additional payload.
PAYLOAD_TYPE_DEVICE_SPECIFIC	9	The buffer payload contains device specific data. The GenTL Producer should report additional information through the corresponding BUFFER_INFO_CMD commands.
PAYLOAD_TYPE_MULTI_PART	10	The buffer payload contains multiple parts of different payload types. Information about the individual parts should be queried using DSGetNumBufferParts and DSGetBufferPartInfo functions.
PAYLOAD_TYPE_GENDC	11	The buffer payload contains a GenDC container. Its contents must be interpreted based on the rules defined in GenDC specification. GenTL specific details related to GenDC payload transfer are defined in 5.7.3 .
PAYLOAD_TYPE_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs which are implementation specific.

6.4.4.6 PIXELFORMAT_NAMESPACE_IDS

```
enum PIXELFORMAT_NAMESPACE_IDS
```

This enumeration defines constants to interpret the pixel formats provided through [BUFFER_INFO_PIXELFORMAT](#).

Enumerator	Value	Description
PIXELFORMAT_NAMESPACE_UNKNOWN	0	The interpretation of the pixel format values is unknown to the GenTL Producer.
PIXELFORMAT_NAMESPACE_GEV	1	The interpretation of the pixel

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Enumerator	Value	Description
		format values is referencing GigE Vision 1.x.
PIXELFORMAT_NAMESPACE_IIDC	2	The interpretation of the pixel format values is referencing IIDC 1.x.
PIXELFORMAT_NAMESPACE_PFNC_16BIT	3	The interpretation of the pixel format values is referencing PFNC 16Bit Values It is recommended to use the PFNC32 namespace when ever possible or even do the translation in the GenTL Producer since the support in GenTL consumers for it is expected to be much broader.
PIXELFORMAT_NAMESPACE_PFNC_32BIT	4	The interpretation of the pixel format values is referencing PFNC 32Bit Values.
PIXELFORMAT_NAMESPACE_CUSTOM_ID	1000	The interpretation of the pixel format values is GenTL Producer specific.

6.4.4.7 PIXELENDIANNESS_IDS

```
enum PIXELENDIANNESS_IDS
```

This enumeration defines constants describing endianness of multi-byte pixel data in a buffer. These values are returned by a call to [DSGetBufferInfo](#) with the command [BUFFER_INFO_PIXELENDIANNESS](#).

Enumerator	Value	Description
PIXELENDIANNESS_UNKNOWN	0	Endianness of the pixel data is unknown to the GenTL Producer.
PIXELENDIANNESS_LITTLE	1	The pixel data is stored in little endian format.
PIXELENDIANNESS_BIG	2	The pixel data is stored in big endian format.

6.4.4.8 STREAM_INFO_CMD

```
enum STREAM_INFO_CMD
```

This enumeration defines commands to retrieve information with the [DSGetInfo](#) function on a data stream handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
STREAM_INFO_ID	M	0	Unique ID of the data stream. Data type: STRING
STREAM_INFO_NUM_DELIVERED	O	1	Number of delivered buffers since last acquisition start. Data type: UINT64
STREAM_INFO_NUM_UNDERRUN	O	2	Number of lost frames due to queue underrun. This number is initialized with zero at the time the stream is opened and incremented every time the data could not be acquired because there was no buffer in the input pool. Data type: UINT64
STREAM_INFO_NUM_ANNOUNCED	O	3	Number of announced buffers. Data type: SIZET
STREAM_INFO_NUM_QUEUED	O	4	Number of buffers in the input pool plus the buffer(s) currently being filled. This does not include the buffers in the output queue. The intention of this informational value is to prevent/early detect an underrun of the acquisition buffers. Data type: SIZET
STREAM_INFO_NUM_AWAIT_DELIVERY	O	5	Number of buffers in the output buffer queue. Data type: SIZET
STREAM_INFO_NUM_STARTED	O	6	Number of frames started in the acquisition engine. This number is incremented every time a new buffer is started to be filled (data written to) regardless if the buffer is later delivered to the user or discarded for any reason. This number is initialized with 0 at the time of the stream is opened. It is not reset until the stream is closed. Data type: UINT64
STREAM_INFO_PAYLOAD_SIZE	CM	7	Size of the expected data in bytes. Data type: SIZET

Enumerator	Impl	Value	Description
STREAM_INFO_IS_GRABBING	M	8	Flag indicating whether the acquisition engine is started or not. This is independent from the acquisition status of the remote device. Data type: BOOL8
STREAM_INFO_DEFINES_PAYLOAD_SIZE	M	9	Flag indicating that this data stream defines a payload size independent from the remote device. If <code>false</code> the size of the expected payload size is to be retrieved from the remote device. If <code>true</code> the expected payload size is to be inquired from the Data Stream module. In case the GenTL Producer does not support this command it is to be interpreted as <code>false</code> . Data type: BOOL8
STREAM_INFO_TLTYPE	M	10	Transport layer technology that is supported. See string constants in chapter 6.6.1. Data type: STRING
STREAM_INFO_NUM_CHUNKS_MAX	CM	11	Maximum number of chunks to be expected in a buffer (can be used to allocate the array for the DSGetBufferChunkData function). In case this is not known a priori by the GenTL Producer the DSGetInfo function returns <code>GC_ERR_NOT_AVAILABLE</code> . This maximum must not change during runtime. Data type: SIZET
STREAM_INFO_BUF_ANNOUNCE_MIN	M	12	Minimum number of buffers to announce. In case this is not known a priori by the GenTL Producer the DSGetInfo function returns a <code>GC_ERR_NOT_AVAILABLE</code> error. This minimum may change during runtime when changing parameters through the node map. Data type: SIZET
STREAM_INFO_BUF_ALIGNMENT	O	13	Alignment size in bytes of the buffer base pointer passed to DSAnnounceBuffer . If a buffer is passed to

Enumerator	Impl	Value	Description
			<p>DSAnnounceBuffer which is not aligned according to the alignment size it is up to the Producer to either reject the buffer and return the GC_ERR_INVALID_BUFFER error code or to cope with a potential overhead and use the unaligned buffer as is. In case there is no special alignment needed the GenTL Producer should report a 1. For composite buffers the same rules apply per-segment. Data type: SIZET</p>
STREAM_INFO_FLOW_TABLE	CM	14	<p>Current state of flow mapping table in GenDC format (if available). Return the GC_ERR_NO_DATA error code if the stream will not use flow mechanism in current configuration. Data type: BUFFER</p>
STREAM_INFO_GENDC_PREFETCH_DESCRIPTOR	O	15	<p>Prefetch version of the GenDC descriptor corresponding to the current stream status, if available and if the stream/device is currently configured to stream in GenDC format. Return the GC_ERR_NO_DATA error code if the stream will not output GenDC payload in current configuration. Data type: BUFFER</p>
STREAM_INFO_CUSTOM_ID	O	1000	<p>Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom STREAM_INFO_COMMANDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.</p>

6.4.4.9 BUFFER_PART_INFO_CMD

enum BUFFER_PART_INFO_CMD

This enumeration defines commands to retrieve information with the [DSGetBufferPartInfo](#) function on a buffer handle. In case a [BUFFER_PART_INFO_CMD](#) is not available or not implemented the [DSGetBufferPartInfo](#) function must return the appropriate error return value.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command even though it might return NI or NA under certain circumstances. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
BUFFER_PART_INFO_BASE	M	0	Base address of the buffer part memory. This is the address where the valid buffer part data start, not considering any padding between data parts or buffer alignment. Data type: PTR
BUFFER_PART_INFO_DATA_SIZE	M	1	Size of the buffer part in bytes. Actual size of the data within this buffer part should be reported. Eventual padding between buffer parts is not included. In case of variable payload type only the size of valid data within the buffer part is reported. Data type: SIZET
BUFFER_PART_INFO_DATA_TYPE	M	2	Type of the data in given part. This information refers to the constants defined in PARTDATATYPE_IDS. Data type: SIZET
BUFFER_PART_INFO_DATA_FORMAT	M	3	Format of the individual items (such as pixels) in the buffer part. The interpretation of the format is specific to every data type (BUFFER_PART_INFO_DATA_TYPE), as specified in definitions of individual PARTDATATYPE_IDS . The actual meaning of the data format

Enumerator	Impl	Value	Description
			depends on the namespace the format belongs to which can be inquired using the <code>BUFFER_PART_INFO_DATA_FORMAT_NAMESPACE</code> command (although for the standard PARTDATATYPE_IDS a recommended data format namespace is always specified). Data type: UINT64
<code>BUFFER_PART_INFO_DATA_FORMAT_NAMESPACE</code>	M	4	This information refers to the constants defined in PIXELFORMAT_NAMESPACE_IDS to allow interpretation of <code>BUFFER_PART_INFO_DATA_FORMAT</code> . Data type: UINT64
<code>BUFFER_PART_INFO_WIDTH</code>	CM	5	Width of the data in the buffer part in number of pixels. If the information is not applicable to given data type, the query should result in <code>GC_ERR_NOT_AVAILABLE</code> . Data type: SIZET
<code>BUFFER_PART_INFO_HEIGHT</code>	CM	6	Height of the data in the buffer part in number of pixels. If the information is not applicable to given data type, the query should result in <code>GC_ERR_NOT_AVAILABLE</code> . Data type: SIZET
<code>BUFFER_PART_INFO_XOFFSET</code>	CM	7	XOffset of the data in the buffer part in number of pixels from the image origin to handle areas of interest. If the information is not applicable to given data type, the query should result in <code>GC_ERR_NOT_AVAILABLE</code> . Data type: SIZET
<code>BUFFER_PART_INFO_YOFFSET</code>	CM	8	YOffset of the data in the buffer part in number of pixels from the image origin

Enumerator	Impl	Value	Description
			<p>to handle areas of interest.</p> <p>If the information is not applicable to given data type, the query should result in GC_ERR_NOT_AVAILABLE.</p> <p>Data type: SIZET</p>
BUFFER_PART_INFO_XPADDING	CM	9	<p>XPadding of the data in the buffer part in number of pixels.</p> <p>If the information is not applicable to given data type, the query should result in GC_ERR_NOT_AVAILABLE.</p> <p>Data type: SIZET</p>
BUFFER_PART_INFO_SOURCE_ID	O	10	<p>Identifier allowing to group data parts belonging to the same source (usually corresponding with the SourceIDValue/ChunkSourceIDValue features from SFNC). Parts marked with the same source ID can be pixel-mapped together. Parts carrying data from different ROI's of the same source would typically be marked with the same source ID.</p> <p>It is not mandatory that source ID's within a given buffer make a contiguous sequence of numbers starting with zero.</p> <p>Note: for example with a dual-source 3D camera, the buffer can contain data parts carrying the 3D data and Confidence data corresponding to both of the two different sources. In this case the source ID helps to match the 3D and Confidence parts belonging together.</p> <p>This information refers for example to the information provided in the GigE Vision image stream multi-part data leader.</p> <p>When given part is not tagged with a specific source ID, the function should</p>

Enumerator	Impl	Value	Description
			return GC_ERR_NOT_AVAILABLE. Data type: UINT64
BUFFER_PART_INFO_DELIVERED_IMAGEHEIGHT	CM	11	The number of lines in the current buffer part as delivered by the transport mechanism. For area scan type images this is usually the number of lines configured in the device. For variable size linescan images this number may be lower than the configured image height. This information refers for example to the information provided in the GigE Vision image stream data trailer. For other technologies, this is to be implemented accordingly. Data type: SIZET
BUFFER_PART_INFO_REGION_ID	O	12	Identifier allowing to group data parts belonging to the same region (usually corresponding with the RegionIDValue/ChunkRegionIDValue features from SFNC). For image based data, all data parts tagged with the same region ID must by definition carry the same region offset/size parameters. It is not mandatory that region ID's within a given buffer make a contiguous sequence of numbers starting with zero. This information refers for example to the information provided in the GigE Vision image stream multi-part data leader. When given part is not tagged with a specific region ID, the function should return GC_ERR_NOT_AVAILABLE. Data type: UINT64
BUFFER_PART_INFO_DATA_PURPOSE_ID	CM	13	Identifier used to tag data parts having the same purpose (usually corresponding with the

Enumerator	Impl	Value	Description
			<p>ComponentIDValue/ ChunkComponentIDValue features from SFNC). The receiver can use this tag to associate parts that convey the same purpose. In many cases, the data purpose is defined by the type of component the data represents. This information refers for example to the information provided in the GigE Vision image stream multi-part data leader.</p> <p>When given part is not tagged with a specific data purpose ID, the function should return GC_ERR_NOT_AVAILABLE. Data type: UINT64</p>
BUFFER_PART_INFO_CUSTOM_ID	O	1000	<p>Starting value for GenTL Producer custom IDs which are implementation specific.</p> <p>If a generic GenTL Consumer is using custom BUFFER PART INFO CMDs provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.</p>

6.4.4.10 PARTDATATYPE_IDS

```
enum PARTDATATYPE_IDS
```

This enumeration defines constants to give a hint on the data type to be expected in the buffer part. These values are returned by a call to [DSGetBufferPartInfo](#) with the command [BUFFER PART INFO DATA TYPE](#). The part data type is intended to describe data in individual parts of a multi-part buffer.

Enumerator	Value	Description
PART_DATATYPE_UNKNOWN	0	The GenTL Producer is not aware of

Enumerator	Value	Description
		the data type of the data in the provided buffer part. From the GenTL Consumer perspective this can be handled as raw data.
PART_DATATYPE_2D_IMAGE	1	Color or monochrome (2D) image. This part carries all the pixel data of given image (even if the image is represented by a single-plane pixel format). It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.
PART_DATATYPE_2D_PLANE_BIPLANAR	2	Single color plane of a planar (2D) image. The data should be linked with the other color planes to get the complete image. The complete image consists of 2 planes. The planes of a given planar image must be placed as consecutive parts within the buffer. It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.
PART_DATATYPE_2D_PLANE_TRIPLANAR	3	Single color plane of a planar (2D) image. The data should be linked with the other color planes to get the complete image. The complete image consists of 3 planes. The planes of a given planar image must be placed as consecutive parts within the buffer. It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.

Enumerator	Value	Description
PART_DATATYPE_2D_PLANE_QUADPLANAR	4	<p>Single color plane of a planar (2D) image. The data should be linked with the other color planes to get the complete image.</p> <p>The complete image consists of 4 planes.</p> <p>The planes of a given planar image must be placed as consecutive parts within the buffer.</p> <p>It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>
PART_DATATYPE_3D_IMAGE	5	<p>3D image (pixel coordinates). This part carries all the pixel data of given image (even if the image is represented by a single-plane pixel format, for example when transferring the depth map only).</p> <p>It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>
PART_DATATYPE_3D_PLANE_BIPLANAR	6	<p>Single plane of a planar 3D image. The data should be linked with the other coordinate planes to get the complete image.</p> <p>The complete image consists of 2 planes.</p> <p>The planes of a given planar image must be placed as consecutive parts within the buffer.</p> <p>It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>
PART_DATATYPE_3D_PLANE_TRIPLANAR	7	<p>Single plane of a planar 3D image. The data should be linked with the other coordinate planes to get the</p>

Enumerator	Value	Description
		<p>complete image. The complete image consists of 3 planes. The planes of a given planar image must be placed as consecutive parts within the buffer. It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>
PART_DATATYPE_3D_PLANE_QUADPLANAR	8	<p>Single plane of a planar 3D image. The data should be linked with the other coordinate planes to get the complete image. The complete image consists of 4 planes. The planes of a given planar image must be placed as consecutive parts within the buffer. It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>
PART_DATATYPE_CONFIDENCE_MAP	9	<p>Confidence of the individual pixel values. Expresses the level of validity of given pixel values. Confidence map is always used together with one or more additional image-based parts matching 1:1 dimension-wise. Each value in the confidence map expresses level of validity of the image pixel at matching position. The data format should be a Confidence PFNC format. It is recommended to use PIXELFORMAT_NAMESPACE_PFN_C_32BIT data format with this data type whenever possible.</p>

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Enumerator	Value	Description
PART_DATATYPE_JPEG	10	JPEG compressed data in the format described in GEV 2.0.
PART_DATATYPE_JPEG2000	11	JPEG 2000 compressed data in the format described in GEV 2.0.
PART_DATATYPE_CUSTOM_ID	1000	Starting value for GenTL Producer custom IDs which are implementation specific.

Note: GigE Vision reports chunk data (when attached to multi-part payload) as an additional “part” for its own stream protocol purposes. However, from a GenTL point of view, the chunk data is an extension of the basic payload, not part of it, therefore the chunk data should not be presented as one of the buffer parts. Therefore also no chunk data specific part datatype ID is introduced in the chapter above.

6.4.4.11 FLOW_INFO_CMD

```
enum FLOW_INFO_CMD
```

This enumeration defines commands to retrieve information with the [DSGetFlowInfo](#) function on a data stream handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
FLOW_INFO_SIZE	M	0	Size of the flow in bytes. This information is essential for GenTL Consumer to be able to allocate suitable buffers for flow acquisition using DSAnnounceCompositeBuffer . In case of GenDC streaming this directly corresponds to the information from GenDC mapping table. Note: querying the flow structure through FLOW_INFO_SIZE and through STREAM_INFO_FLOW_TABLE must yield same results. Data type: SIZET
FLOW_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific.

Enumerator	Impl	Value	Description
			If a generic GenTL Consumer is using custom FLOW_INFO_CMDs provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.4.12 SEGMENT_INFO_CMD

```
enum SEGMENT_INFO_CMD
```

This enumeration defines commands to retrieve information with the [DSGetBufferSegmentInfo](#) function on a buffer handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
SEGMENT_INFO_BASE	M	0	Base address of the buffer segment memory as passed to the DSAnnounceCompositeBuffer function. This is also the address where the payload within the segment starts. Data type: PTR
SEGMENT_INFO_SIZE	M	1	Size of the buffer segment in bytes as passed to the DSAnnounceCompositeBuffer function. Data type: SIZET
SEGMENT_INFO_IS_INCOMPLETE	O	2	Flag to indicate that the buffer segment was filled but an error occurred during that process. For technologies or use cases where this is difficult to track, it is valid leave the command not implemented. Data type: BOOL8
SEGMENT_INFO_SIZE_FILLED	O	3	Number of bytes written into the buffer segment the last time it has been filled. This value is reset to 0 when the buffer

Enumerator	Impl	Value	Description
			is placed into the Input Buffer Pool. Used similar way as BUFFER_INFO_SIZE_FILLED at buffer level. For technologies or use cases where this is difficult to track, it is valid leave the command not implemented. Data type: SIZET
SEGMENT_INFO_DATA_SIZE	O	4	Size of the data intended to be written to the buffer last time it has been filled. This value is reset to 0 when the buffer is placed into the Input Buffer Pool. Used similar way as BUFFER_INFO_DATA_SIZE at buffer level. For technologies or use cases where this is difficult to track, it is valid leave the command not implemented. Data type: SIZET
SEGMENT_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom SEGMENT_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.5 Port Enumerations

6.4.5.1 PORT_INFO_CMD

```
enum PORT_INFO_CMD
```

This enumeration defines commands to retrieve information with the [GCGetPortInfo](#) function on a module or remote device handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command even though it might return NI or NA under certain circumstances. Optional means that it is up to the implementor if a given

command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
PORT_INFO_ID	M	0	<p>Unique ID of the module the port references.</p> <p>In case of the remote device module PORT_INFO_ID returns the same ID as for the local device module.</p> <p>In case of a buffer PORT_INFO_ID returns the address of the buffer as hex string without the leading '0x'. For composite buffers the address of the first segment is used.</p> <p>Data type: STRING</p>
PORT_INFO_VENDOR	M	1	<p>Port vendor name.</p> <p>In case the underlying module has no explicit vendor the vendor of the GenTL Producer is to be used. In case of a Buffer or a Data Stream the GenTL Producer vendor and model are to be used.</p> <p>Data type: STRING</p>
PORT_INFO_MODEL	M	2	<p>Port model name.</p> <p>The port model references the model of the underlying module. For example if the port is for the configuration of a TLSystem module the PORT_INFO_MODEL returns the model of the TLSystem Module.</p> <p>In case the underlying module has no explicit model, the model of the GenTL Producer is to be used. So in case of a Buffer or a Data Stream the GenTL Producer model is to be used.</p> <p>Data type: STRING</p>
PORT_INFO_TLTYPE	M	3	<p>Transport layer technology that is supported. See string constants in chapter 6.6.1.</p> <p>Data type: STRING</p>
PORT_INFO_MODULE	M	4	<p>GenTL Module the port refers to:</p> <ul style="list-style-type: none"> • "TLSystem" for the System module. • "TLInterface" for the Interface module. • "TLDevice" for the Device module.

Enumerator	Impl	Value	Description
			<ul style="list-style-type: none"> “TLDataStream” for the Data Stream module. “TLBuffer” for the Buffer module. “Device” for the remote device. Data type: STRING
PORT_INFO_LITTLE_ENDIAN	M	5	Flag indicating that the port’s data is little endian. Data type: BOOL8
PORT_INFO_BIG_ENDIAN	M	6	Flag indicating that the port’s data is big endian. Data type: BOOL8
PORT_INFO_ACCESS_READ	M	7	Flag indicating that read access is allowed. Data type: BOOL8
PORT_INFO_ACCESS_WRITE	M	8	Flag indicating that write access is allowed. Data type: BOOL8
PORT_INFO_ACCESS_NA	M	9	Flag indicating that the port is currently not available. Data type: BOOL8
PORT_INFO_ACCESS_NI	M	10	Flag indicating that no port is implemented. This is only valid on the Buffer module since on all other modules the port is mandatory. Data type: BOOL8
PORT_INFO_VERSION	M	11	Version of the port. Data type: STRING
PORT_INFO_PORTNAME	M	12	Name of the port as referenced in the XML description. This name is used to connect this port to the nodemap instance of this module. Data type: STRING
PORT_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom PORT_INFO_CMDs provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.5.2 URL_INFO_CMD

```
enum URL_INFO_CMD
```

This enumeration defines commands to retrieve information with the [GCGetPortURLInfo](#) function on a module or remote device handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command even though it might return NA. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
URL_INFO_URL	M	0	URL as defined in chapter 4.1.2 Data type: STRING
URL_INFO_SCHEMA_VERSION_MAJOR	CM	1	Major version of the schema this URL refers to. Data type: INT32
URL_INFO_SCHEMA_VERSION_MINOR	CM	2	Minor version of the schema this URL refers to. Data type: INT32
URL_INFO_FILE_VERSION_MAJOR	CM	3	Major version of the XML-file this URL refers to. Data type: INT32
URL_INFO_FILE_VERSION_MINOR	CM	4	Minor version of the XML-file this URL refers to. Data type: INT32
URL_INFO_FILE_VERSION_SUBMINOR	CM	5	Subminor version of the XML-file this URL refers to. Data type: INT32
URL_INFO_FILE_SHA1_HASH	CM	6	SHA1 Hash of the XML-file this URL refers to. The size of the provided buffer is 160Bit according to the SHA1 specification. Data type: BUFFER
URL_INFO_FILE_REGISTER_ADDRESS	CM	7	Register address of the XML-File in the device’s register map. In case the XML is not locally stored in the device’s register map the info function should return a <code>GC_ERR_NOT_AVAILABLE</code> . Data type: UINT64

Enumerator	Impl	Value	Description
URL_INFO_FILE_SIZE	CM	8	File size of the XML-File in bytes in the register map. For URLs starting with 'file:' or 'http:' the according info function should return a GC_ERR_NOT_AVAILABLE. Data type: UINT64
URL_INFO_SCHEME	CM	9	Scheme of the URL. Possible values are defined in URL_SCHEME_IDS . Data type: INT32
URL_INFO_FILENAME	CM	10	Filename in case the scheme of the URL is URL_SCHEME_FILE or as a hint if the scheme is URL_SCHEME_LOCAL . Data type: STRING
URL_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom URL_INFO_COMMANDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.5.3 URL_SCHEME_IDS

```
enum URL_SCHEME_IDS
```

This enumeration defines the values to be retrieved through a call to [GCGetPortURLInfo](#) with the command [URL_INFO_SCHEME](#).

Enumerator	Value	Description
URL_SCHEME_LOCAL	0	The XML-File is to be retrieved from the local register map. The address and size where it can be read can be queried using the GCGetPortURLInfo function with the URL_INFO_FILE_REGISTER_ADDRESS and URL_INFO_FILE_SIZE command.

URL_SCHEME_HTTP	1	The XML-file can be retrieved from a webserver using the http protocol. The URL where it can be downloaded can be queried using the GCGetPortURLInfo function with the URL INFO URL command.
URL_SCHEME_FILE	2	The XML-file can be read from the local hard disk. The filename can be queried through GCGetPortURLInfo function using the URL INFO FILENAME command.
URL_SCHEME_CUSTOM_ID	1000	Starting value for custom IDs which are implementation specific.

6.4.6 Signaling Enumerations

6.4.6.1 EVENT_DATA_INFO_CMD

```
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).

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command even though it might return NI or NA under certain circumstances. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
EVENT_DATA_ID	M	0	Attribute 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	M	1	Defines additional data to an ID. This can be, e.g., the error message for an error event.
EVENT_DATA_NUMID	M	2	Attribute in the event data to identify the object or feature the event refers to. It is the numeric representation of EVENT_DATA_ID if applicable. In case it is not possible to map EVENT_DATA_ID to a number the EventGetDataInfo function

Enumerator	Impl	Value	Description
			returns GC_ERR_NOT_AVAILABLE. Data type: UINT64
EVENT_DATA_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom EVENT_DATA_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.6.2 EVENT_INFO_CMD

```
enum EVENT_INFO_CMD
```

This enumeration defines command to retrieve information with the [EventGetInfo](#) function on an event handle.

The column labeled “Impl” in the following table lists if the implementation of a given command is mandatory (M), optional (O) or conditional mandatory (CM). Mandatory means that a GenTL Producer must implement the listed command even though it might return NI or NA under certain circumstances. Optional means that it is up to the implementor if a given command is implemented or not. Conditional Mandatory means that command is to be implemented if possible.

Enumerator	Impl	Value	Description
EVENT_EVENT_TYPE	M	0	The event type of the event handle. Data type: INT32 (EVENT_TYPE enum value).
EVENT_NUM_IN_QUEUE	M	1	Number of events in the event data queue. Data type: SIZET
EVENT_NUM_FIRED	O	2	Number of events that were fired since the registration of the event through a call to GCRegisterEvent . A fired event is either still in the internal queue or already delivered to the user or discarded through EventFlush . Data type: UINT64
EVENT_SIZE_MAX	M	3	Maximum size in bytes of the event data provided by the event. In case this

Enumerator	Impl	Value	Description
			is not known a priori by the GenTL Producer the EventGetInfo function returns GC_ERR_NOT_AVAILABLE. This maximum size must not change during runtime. Data type: SIZET
EVENT_INFO_DATA_SIZE_MAX	M	4	Maximum size in bytes of the information output buffer of EventGetDataInfo function for EVENT_DATA_VALUE . In case this is not known a priori by the GenTL Producer the EventGetDataInfo function returns the GC_ERR_NOT_AVAILABLE error. This maximum size must not change during runtime. Data type: SIZET
EVENT_INFO_CUSTOM_ID	O	1000	Starting value for GenTL Producer custom IDs which are implementation specific. If a generic GenTL Consumer is using custom EVENT_INFO_CMDS provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.4.6.3 EVENT_TYPE

```
enum EVENT_TYPE
```

Known event types that can be registered on certain modules with the [GCRegisterEvent](#) function. See chapter [4.2 Signaling page 35](#) 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:

Enumerator	Value	Description
		<ul style="list-style-type: none"> EVENT_DATA_ID Data type: INT32 (GC_ERROR) EVENT_DATA_VALUE Data type: STRING (Description)
EVENT_NEW_BUFFER	1	<p>Notification on newly filled buffers. Values that can be retrieved are:</p> <ul style="list-style-type: none"> EVENT_DATA_ID Data type: PTR (Buffer handle) EVENT_DATA_VALUE Data type: PTR (Private pointer)
EVENT_FEATURE_INVALIDATE	2	<p>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:</p> <ul style="list-style-type: none"> EVENT_DATA_ID Data type: STRING (Feature name)
EVENT_FEATURE_CHANGE	3	<p>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:</p> <ul style="list-style-type: none"> EVENT_DATA_ID Data type: STRING (Feature name) EVENT_DATA_VALUE Data type: STRING (Feature value)
EVENT_REMOTE_DEVICE	4	<p>Notification if the GenTL Producer wants to inform the GenICam GenApi instance of the remote device that a GenApi compatible event was fired. This Event is to be registered on a Local Device module.</p> <p>Values that can be retrieved are:</p> <ul style="list-style-type: none"> EVENT_DATA_ID String representation of the EventID number in hexadecimal numbers with even number of digits and without the leading '0x'. Data type: STRING (Event ID) EVENT_DATA_VALUE Corresponds to the data addressable through the remote device's nodemap event port, beginning of the buffer corresponding to address 0. Data type: BUFFER (optional data)

Enumerator	Value	Description
		This event type used to be called <code>EVENT_FEATURE_DEVEVENT</code> but has been renamed for a more intuitive understanding.
<code>EVENT_MODULE</code>	5	Notification that one GenTL Producer module wants to inform the GenICam GenApi instance of this module that a GenApi compatible event was fired. This Event is to be registered on any module handle except on the Remote Device. Values that can be retrieved are: <ul style="list-style-type: none"> <code>EVENT_DATA_ID</code> String representation of the EventID number in hexadecimal numbers with even number of digits and without the leading '0x'. Data type: STRING (Event ID) <code>EVENT_DATA_VALUE</code> Corresponds to the data addressable through the module's nodemap event port, beginning of the buffer corresponding to address 0. Data type: BUFFER (optional data)
<code>EVENT_CUSTOM_ID</code>	1000	Starting value for GenTL Producer custom events which are implementation specific. If a generic GenTL Consumer is using custom <code>EVENT_TYPES</code> provided through a specific GenTL Producer implementation it must differentiate the handling of different GenTL Producer implementations in case other implementations will not provide that custom id or will use a different meaning with it.

6.5 Structures

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

6.5.1 Data Stream Structures

6.5.1.1 SINGLE_CHUNK_DATA

```
struct SINGLE_CHUNK_DATA
```

Layout of the array elements being used in the function [DSGetBufferChunkData](#) to carry information about individual chunks present in the buffer.

Member	Type	Description
ChunkID	uint64_t	Numeric representation of the chunk's ChunkID.
ChunkOffset	ptrdiff_t	Offset of the chunk's data from the start of the buffer (in bytes).
ChunkLength	size_t	Size of the given chunk data (in bytes).

6.5.1.2 DS_BUFFER_INFO_STACKED

```
struct DS_BUFFER_INFO_STACKED
```

Layout of the array elements being used in the function [DSGetBufferInfoStacked](#) to carry information about multiple buffer infos as defined in [BUFFER_INFO_CMD](#).

Member	Type	Description
iInfoCmd [in]	BUFFER_INFO_CMD	The buffer info to be retrieved.
iType [out]	INFO_DATATYPE	Data type of the <i>pBuffer</i> content as defined in the BUFFER_INFO_CMD and INFO_DATATYPE .
pBuffer [in,out]	void*	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>iSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>iType</i> is a string the size includes the terminating 0.
iSize [in,out]	size_t	<i>pBuffer</i> equal NULL: out: minimal size of <i>pBuffer</i> in bytes to hold all information. <i>pBuffer</i> unequal NULL: in: size of the provided <i>pBuffer</i> in bytes. out: number of bytes filled by the function.
iResult [out]	GC_ERROR	The result of the buffer info query.

[DSGetBufferInfoStacked](#) queries multiple buffer infos as defined in [BUFFER_INFO_CMD](#) at once.

The purpose and “direction” (in/out) of the structure data members is same as corresponding parameters of the [DSGetBufferInfo](#) function. Similar as other get-info functions, the buffer size required to hold given information can be negotiated first as described for the `pBuffer==NULL` above.

When retrieving multiple infos through [DSGetBufferInfoStacked](#) call, each `DS_BUFFER_INFO_STACKED` structure is handled independently on the others. GenTL Consumer should receive identical output (*iResult* and the value in *pBuffer*), no matter if it used single [DSGetBufferInfoStacked](#) or sequence of [DSGetBufferInfo](#) calls.

Each *iResult* member of the many DS_BUFFER_INFO_STACKED structures represents the result of exactly one buffer info query. The applicable error codes of the [DSGetBufferInfo](#) function are valid for *iResult*. These are listed below:

- GC_ERR_SUCCESS Operation was successful; no error occurred.
- GC_ERR_NOT_IMPLEMENTED Specified *iInfoCmd* is not implemented.
- GC_ERR_BUFFER_TOO_SMALL *pBuffer* is not NULL and the value of *iSize* is too small to receive the expected amount of data.
- GC_ERR_NOT_AVAILABLE The request is implemented but the requested information is currently not available for any reason.

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

6.5.1.3 DS_BUFFER_PART_INFO_STACKED

```
struct DS_BUFFER_PART_INFO_STACKED
```

Layout of the array elements being used in the function [DSGetBufferPartInfoStacked](#) to carry information about multiple buffer part infos as defined in [BUFFER PART INFO CMD](#).

Member	Type	Description
<i>iPartIndex</i> [in]	uint32_t	Zero based index of the buffer part to query.
<i>iInfoCmd</i> [in]	BUFFER PART INFO CMD	The buffer part info to be retrieved.
<i>iType</i> [out]	INFO DATATYPE	Data type of the <i>pBuffer</i> content as defined in the BUFFER PART INFO CMD and INFO DATATYPE .
<i>iResult</i> [out]	GC ERROR	The result of the buffer part info query.
<i>pBuffer</i> [in,out]	void*	Pointer to a user allocated buffer to receive the requested information. If this parameter is NULL, <i>iSize</i> will contain the minimal size of <i>pBuffer</i> in bytes. If the <i>iType</i> is a string the size includes the terminating 0.
<i>iSize</i> [in,out]	size_t	<i>pBuffer</i> equal NULL: out: minimal size of <i>pBuffer</i> in bytes to hold all information. <i>pBuffer</i> unequal NULL: in: size of the provided <i>pBuffer</i> in bytes. out: number of bytes filled by the function.

[DSGetBufferPartInfoStacked](#) queries multiple buffer part infos as defined in [BUFFER PART INFO CMD](#) at once.

The purpose and “direction” (in/out) of the structure data members is same as corresponding parameters of the [DSGetBufferPartInfo](#) function. Similar as other get-info functions,

the buffer size required to hold given information can be negotiated first as described for the `pBuffer==NULL` above.

When retrieving multiple infos through [DSGetBufferPartInfoStacked](#) call, each `DS_BUFFER_PART_INFO_STACKED` structure is handled independently on the others. GenTL Consumer should receive identical output (*iResult* and the value in *pBuffer*), no matter if it used single [DSGetBufferPartInfoStacked](#) or sequence of [DSGetBufferPartInfo](#) calls.

Each *iResult* member of the many `DS_BUFFER_PART_INFO_STACKED` structures represents the result of exactly one buffer part info query. The applicable error codes of the [DSGetBufferPartInfo](#) function are valid for *iResult*. These are listed below:

<code>GC_ERR_SUCCESS</code>	Operation was successful; no error occurred.
<code>GC_ERR_NOT_IMPLEMENTED</code>	Specified <i>iInfoCmd</i> is not implemented.
<code>GC_ERR_BUFFER_TOO_SMALL</code>	<i>pBuffer</i> is not NULL and the value of <i>iSize</i> is too small to receive the expected amount of data.
<code>GC_ERR_NOT_AVAILABLE</code>	The request is implemented but the requested information is currently not available for any reason.
<code>GC_ERR_INVALID_INDEX:</code>	<i>iPartIndex</i> is greater than the number of available buffer parts - 1 retrieved through a call to DSGetNumBufferParts .

Error cases not covered in the list above may return error codes according to chapter [6.1.5 Error Handling](#) on page [61](#).

(Note: The order of struct members in DS_BUFFER_PART_INFO_STACKED vs. DS_BUFFER_INFO_STACKED is intentionally slightly different to keep both structs naturally aligned.)

6.5.2 Signaling Structures

6.5.2.1 EVENT_NEW_BUFFER_DATA

```
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.

6.5.3 Port Structures

6.5.3.1 PORT_REGISTER_STACK_ENTRY

```
struct PORT_REGISTER_STACK_ENTRY
```

Layout of the array elements being used in the function [GCWritePortStacked](#) and [GCReadPortStacked](#) to accomplish a stacked register read/write operations.

Member	Type	Description
Address	uint64_t	Register address
Buffer	void *	Pointer to the buffer receiving the data being read/containing the data to write.
Size	size_t	Number of bytes to read / write. The provided <i>Buffer</i> must be at least that size.

6.6 String Constants

6.6.1 Transport Layer Types

String constants for transport layer technologies that are supported to be used with the module info commands xxx_INFO_TLTYPE (for example TL_INFO_TLTYPE) inquiry commands.

Transport Technology Standard	String Constant
GigE Vision	“GEV”
Camera Link	“CL”
IIDC 1394	“IIDC”
USB video class	“UVC”
CoaXPress	“CXP”
Camera Link HS	“CLHS”
USB3 Vision Standard	“U3V”
Generic Ethernet	“Ethernet”
PCI / PCIe	“PCI”
Mixed	“Mixed” This type is only valid for the System module in case the different Interface modules with a single system are of different types. All other modules must be of a defined type.
Non standard transport technology, not covered by other constants.	”Custom“

GEN<i>CAM		
Version 1.6	GenTL Standard	

Transport Technology Standard	String Constant

6.7 Numeric Constants

Numerical constants used in the GenTL Producer API.

Constant	Value(Type)	Description
GENTL_INVALID_HANDLE	NULL (void *)	This value is indicating an invalid handle for any handle type used within GenTL Producer API.
GENTL_INFINITE	0xFFFFFFFFFFFFFFFF (UINT64)	Value indicating an infinite number (e.g., timeout or number of buffers) to be used with API functions.

7 Standard Features 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 Features Naming Convention document (SFNC) and in the GenTL Standard Features Naming Convention document (GenTL SFNC).

For technical reasons the different transport layer technologies and protocols (e.g., GigE Vision, IIDC 1394, Camera Link, etc.) 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. Mixed-type GenTL Producers must implement mandatory features of all supported technologies in the System node map. The mandatory technology specific features falling under the “InterfaceSelector” might be marked not-available (NA) when an interface implementing other technology is currently selected.

Interface, Device, Data Stream and Buffer node maps are unequivocally bound to a particular transfer technology and thus they must implement only technology specific features of the corresponding technology.

When updating features which are related to information covered also in the C interface it might happen that the data the node map refers to changes unexpectedly. Therefore these values should not be cached in the nodemap but read every time from the module. This especially applies to features under a module selector.

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 7-5: System module information features

Name	Interface	Access	Description
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 ID identifying a GenTL Producer. For example the filename of the GenTL Producer implementation

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Name	Interface	Access	Description
			including its path.
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. See chapter 6.6.1 for possible values.
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 7-6: Interface enumeration features

Name	Interface	Access	Description
InterfaceUpdateList	ICommand	(R)/W	Updates the internal interface list. This feature should be readable if the execution cannot be 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. The selector is 0 based in order to match the index of the C interface.
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 7-7: Interface information features

Name	Interface	Access	Description
------	-----------	--------	-------------

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Name	Interface	Access	Description
InterfaceID	IString	R	GenTL Producer wide unique identifier of the selected interface.
InterfaceType	IEnumeration	R	Identifies the transport layer technology of the interface. See chapter 6.6.1 for possible values.

Table 7-8: Device enumeration features

Name	Interface	Access	Description
DeviceUpdateList	ICommand	(R)/W	Updates the internal device list. This feature should be readable if the execution cannot be performed immediately. The command then returns and the status can be polled. This function interacts with the IUpdateDeviceList 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”. The selector is 0 based in order to match the index of the C interface.
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	Returns the device's access status. Possible values are: <ul style="list-style-type: none"> • “ReadWrite” The device is available to be opened with full access. As soon as the device is open the value should change to “OpenReadWrite” or “OpenRead” Corresponds to DEVICE_ACCESS_STATUS_READWRITE

Name	Interface	Access	Description
			<p>DWRITE.</p> <ul style="list-style-type: none"> • “ReadOnly” The device is available to be opened with read-only access. As soon as the device is open the value should change to “OpenRead”. Corresponds to DEVICE_ACCESS_STATUS_READ_ONLY. • “NoAccess” The device is seen by the producer but not reachable. Corresponds to DEVICE_ACCESS_STATUS_NOACCESS. • “Busy” The device is already opened by another entity. Corresponds to DEVICE_ACCESS_STATUS_BUSY. • “OpenReadWrite” The device is already open by this GenTL Producer with RW access. Corresponds to DEVICE_ACCESS_STATUS_OPEN_READWRITE. • “OpenReadOnly” The device is already opened by this GenTL Producer with RO access. Corresponds to DEVICE_ACCESS_STATUS_OPEN_READONLY.

7.1.3 Device Module

The 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 7-9: Device information features

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Name	Interface	Access	Description
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. See chapter 6.6.1 for possible values.

Table 7-10: Stream enumeration features

Name	Interface	Access	Description
StreamSelector	IInteger	R/W	Selector for the different stream channels. The selector is 0 based in order to match the index of the C interface.
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 7-11: Data Stream information features

Name	Interface	Access	Description
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. This value is volatile. It may change if additional buffers are announced and/or buffers are revoked by the GenTL Consumer.
StreamAcquisitionModeSelector	IEnumeration	R/W	Available buffer handling modes of this Stream. Deprecated. Use “StreamBufferHandlingMode” instead. Value: “Default” (see chapter 5 Acquisition Engine page 42ff)
StreamBufferHandlingMode	IEnumeration	R/W	Available buffer handling modes of this Stream. Value: “Default” (see chapter 5 Acquisition Engine page 42ff)
StreamAnnounceBufferMinimum	IInteger	R	Minimal number of buffers to announce to enable selected buffer handling mode.

GEN<i>i</i>CAM		
Version 1.6	GenTL Standard	

Name	Interface	Access	Description
StreamType	IEnumeration	R	Identifies the transport layer technology of the stream. See chapter 6.6.1 for possible values.

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 7-12: Buffer information features

Name	Interface	Access	Description
BufferData	IRegister	R/(W)	Entire buffer data.
BufferUserData	IInteger	R	Pointer to user data (<i>pPrivate</i>) casted to an integer number referencing GenTL Consumer specific data. It is reflecting the pointer provided by the user data pointer (<i>pPrivate</i>) at buffer announcement. (see chapter 6.3.5 Data Stream Functions page 89ff). This allows the GenTL Consumer to attach information to a buffer.