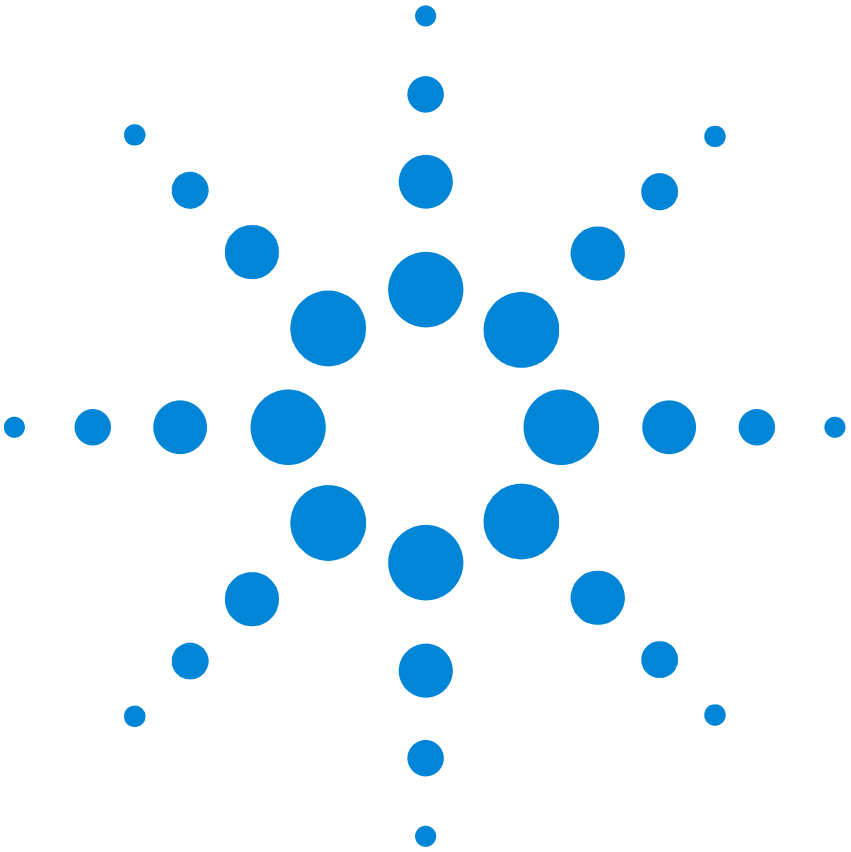


Agilent Technologies E2950 Series InfiniBand Exciser

API Reference

Agilent Technologies

Important Notice

Manual Part Number

5988-5061EN

Revision

Revision 2.0, June 2002

Printed in Germany

Agilent Technologies,
Deutschland GmbH
Herrenberger Str. 130
71034 Boeblingen, Germany

Warranty

The material contained in this document is provided “as is,” and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as “Commercial computer software” as defined in DFAR 252.227-7014 (June 1995), or as a “commercial item” as defined in FAR 2.101(a) or as “Restricted computer software” as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies’ standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Table of Contents

Programming the E2953A/E2954A

This chapter briefly explains the basic ideas behind the programming model of the E2953A/E2954A. The concept is explained for the C++ interface. The TCL interface is built in a similar way.

The main programming interface to the E2953A/E2954A is based on the C++ programming language. This C++ interface can also be accessed from a TCL shell to provide the capabilities of a script language. The TCL commands are completely based on the C++ calls. Both interfaces can be used to integrate the E2953A/E2954A into various test environments and third party test software.

Packet Handling Concept

For every E2953A/E2954A InfiniBand generator in use, you need to create one instance of the class `IGCGenerator`. This class provides all the necessary functions to interact with the generator. It also provides the entire status information and controls the InfiniBand link. The InfiniBand `portinfo` struct is managed here too.

For every E2953A/E2954A generator that is connected, there can be only one controlling generator object. However, you are free to create as many generator objects as there are devices connected to the controlling PC. By default, a newly created generator class is in off-line mode. It needs to get connected to a real device before it can provide all of its functionality.

Basically, every class with property values has the methods `Set` and `Get` to allow it to write and read these properties respectively. Enum values belonging to the appropriate class identify the properties. For a detailed

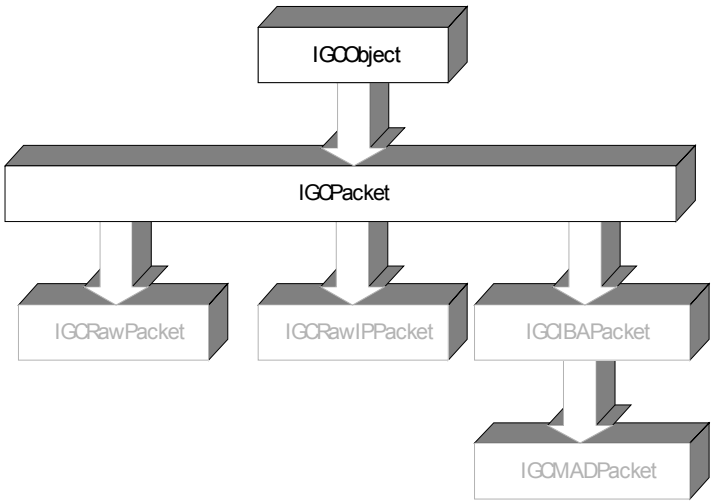
list of these properties refer to *"Properties and Programmatic Settings" on page 4-1*.

Sending Packets

In order to send out packets, you have to create objects of various packet classes and set the internal properties of these packet objects as desired. There are a number of different classes that derive from `IGCPacket`. These can be selected by yourself depending on the type of packet that should be sent. The following figure shows the relationship between the different packet classes.

All types of packets are derived from the class `IGCObject` which implements the set and get functionality for these classes. Using the `Set()` and `Get()` functions you can manipulate packet properties.

Figure 1 Class Architecture of the IGCPacket Package



When an InfiniBand packet is to be created, you first need to set the opcode and the packet type (local or global) in order to obtain access to all the other packet properties. The class `IGCIBAPacket` provides the method `Init()` for this purpose.

To simplify setting up the packet, you can use the generator class to initialize the packet with all the information that is held in the `portinfo`

struct within the generator. The generator provides the method `PacketInit()` for that purpose.

Having created an InfiniBand packet (or a raw packet), you can send it out in one of two ways:

- Direct send

  Using the method `PacketSend()`, you can directly pass an object of the type `IGCPacket` (or derived from `IGCPacket`) to the generator class which then immediately sends out the packet.

- Memory based packet send

  Using the method `TransmitProg()`, you can program up to 512 Kbytes for the E2953A and up to 2048 Kbytes of packets in the transmit memory. Use the call `TransmitRun()`to start the transfer. You can repeat the programmed packet sequence if required.

By mixing these two methods you can create different test scenarios. For example, you can flood the network with a large amount of low priority packets and then insert high priority packets once in a while to check that all participating switches and routers are capable of handling the priorities correctly.

The `IGCPacket` object can be used several times and can even be sent out from different generators. Using the method `AppendBuffer()` and `NewPacket()` a packet can be transformed into a byte stream or vice versa.

Receiving Packets

The E2953A/E2954A generator receives packets in two different ways. They represent default behavior:

- MAD packets (SMD) are stored in an extra FIFO that is exclusively reserved for this type of packet.
- Standard InfiniBand packets are stored in the receive memory. Depending on the mode of the generator, the hardware either controls the incoming packets via flow control packets or takes all packets without checking if they have been picked up by the software (data sink mode).

In order to handle packets you need to register a packet handler with the generator class. This involves deriving a class from the class `IGCPacketHandler` and writing the two methods `CheckPacket()` and `HandlePacket()`. These exist as purely virtual methods in `IGCPacketHandler`. `CheckPacket()` gets called to determine whether the packet handler should deal with the packet. Having done this, the function returns. Subsequently, the base class `IGCPacketHandler` manages the call to Handle`Packet()`.

An additional class derived from `IGCPacketHandler` is included with the product:

- `IGCPacketHandlerTcl` provides a class that can handle tcl scripts. This allows a tcl script to handle incoming packets, simplifying this task.

NOTE Subnet management is implemented as an SMA (subnet management agent), programmed as a tcl script (refer to *"TCL Interface" on page 1-9*).

Exception and Error Handling

This section shows the error mechanisms implemented by the C++ and the TCL interfaces.

Error Mechanisms for the C++ Interface

The following code block shows an example of how to use the exception handling with the API.

```
try
{
  IGCGenerator myGenerator;
  myGenerator.Connect(0);
  myGenerator.Foo();
}
catch (IGCError err)
{
  // Error occurred in try block
  // Do error handling here, e.g. print error message:
  cerr << "Error occurred: " << err;
}
```

To throw an error, use a line similar to the one below:

```
throw(IGCError(IGCError::IGE_FATAL, "Cannot close device"));
```

See also the descriptions in *"Methods of the IGCPacketHandler Class" on page 2-105* and *"EErrtype" on page 3-1*.

Error Mechanism for the TCL Interface

The following example script shows how an E2953A/E2954A gets connected using the TCL script language and shows the error mechanism in case the connect was not successful.

```
if { [catch {
   set portnum 0
   set gen [new_IGCGenerator]
   IGCGenerator_Connect $gen $portnum
   } result]} {
  # error while establishing connection
  puts stderr "ERROR: Cannot connect to generator at port $portnum:
$result"
  } else {
  puts "Connected on port $portnum"
}
```

In order to throw an error, use the following script command.

```
error "Fatal error: Cannot <do whatever the task was>"
```

The `error` command terminates the script unless the error is caught by a `catch` command. Errors from the class `igapi` are automatically caught, that is, the error message is printed to stderr (interactive mode) and the corresponding tcl function returns with `TCL_ERROR`.

Performance Measurement

The performance measurement counts values such as the size of payload, the number of good and bad packets and the number of link packets received and transmitted by the exerciser. The two performance counters that hold the result of the performance measurement are implemented in the hardware of the E2953A/E2954A and are accessed via the `IGCPerfromance` class of the API (see *"IGCPerformance Class" on page 2-12*). For each of the two performance counters, you can separately determine which VLs you want to monitor for incoming and outgoing packets.

When reading out the measurement, you obtain the values that accumulated from the last time you read out the values or started the performance measurement.

Link Packet and Protocol Observer

The API comes with two classes that allow you to observe link packets received by the generator:

- IGCProtocolObserver

  This class lets you get the status of the protocol observer. See *"IGCProtocolObserver Class" on page 2-13* for details.

- IGCLinkPacketStatus

  This class lets you get the status of the link packet itself. See *"IGCLinkPacketStatus Class" on page 2-14* for details.

Both classes just provide "containers" for the information given about the status of the protocol observer and link packets respectively. Controlling, that is:

- Starting and stopping the recording of link packet
- Resetting the protocol observer
- Reading out of the results

is performed via new methods in the generator class. Refer to *"Methods of the IGCGenerator Class" on page 2-18* for more information.

Control Command Language

The basis for the control language is the C++ interface.

TCL Interface

The entire functionality of the InfiniBand Generator can also be accessed via a TCL interface. A part of the software installation is a subnet management agent (SMA) programmed as a TCL script. The SMA handles all incoming MAD packets and registers the generator correctly within the InfiniBand network.

Examples

When you create and use a new class object in TCL it you have to follow the syntax scheme as described in the following table. It is your responsibility to write a catch handler to handle any errors that may occur when a TCL script runs. Failure to do this causes the TCL interpreter to generate an exception.

Table 1 Programming Scheme using TCL

What you intend	What the SW does	Tcl Syntax
Create a new class object in TCL.	Assigns an object of type `classname` to the variable `var`.	`set var [ new_<classname> ?parameter? ]`
Use a class method.	Assumes that `var` contains an object of type `classname`.	`<classname>_<methodname> $var ?parameters?`
Delete an object of type `classname`. Equivalent of calling the destructor.	Assumes that `var` contains an object of type classname.	`delete_<classname> $var`
Set a property to a specific value. Properties are defined in the context of the class in which they are used.	All classes that contain properties are derived from `IGCObject` which implements the `Set()` and `Get()` functions.	`IGCObject_Set $var $<classname>_<propertyname> <value>`

What you intend	What the SW does	Tcl Syntax
Get the value of a property.	Assumes that `val` contains the value and `var` contains a class object.	`Set val [ IGCObject_Get $var $<classname>_<propname> ]`
Create a buffer.	Makes a new buffer to fill with packet bytes later on.	`set buf [new_IGCBuffer]`
Fill the buffer. This appends the packet to the buffer `buf`.	Assumes that `$pkt` contains the packet and `$buf` contains the buffer.	`IGCPacket_AppendBuffer $pkt $buf`
Pop a few bits out of a buffer into a tcl variable.	Assumes that `$buf` contains the buffer.	`set var [IGCBuffer_Pop $buf <lengthinbit>]`
Get a few bits out of a buffer at a certain position	Assumes that `$buf` contains the buffer.	`set var [IGCBuffer_GetAt $buf <offset> <lengthinbit>]`
Push a few bits into a buffer.	Assumes that `$buf` contains the buffer.	`IGCBuffer_Push $buf <lengthinbits> <value>`
Set a few bits into a buffer at a certain position.	Assumes that `$buf` contains the buffer.	`IGCBuffer_SetAt $buf <offset> <lenghtinbits> <value>`

Print the status on the screen.	NOTE: All `Print()` functions are mapped into Tcl to return the printed string. No input parameter has to be specified!	`set stat [new_IGCStatus]`
		
		`IGCGenerator_UpdateStatus $gen $stat`
		
		`set str [IGCStatus_Print $stat]`
		
		`puts "Result: $str"`

NOTE Packets ready and waiting in a buffer may not contain CRC values!

By using these functions, you can also access the buffer bitwise. The class `IGCBuffer` can also be saved to file and restored from file.

Installed TCL Sample Scripts

The following tables describes the scripts installed with the E2953A/E2954A software.

Table 2 Sample scripts installed with the E2953A/E2954A software

Script name	Description
GettingStarted	Starts a rudimentary Subnet Management agent. This script shows how to generate and respond to incoming SMP packets in TCL
Packetbounce	Bounces a packet between two generators
Showprops	Show/Update a window showing all props of an `IGCObject` instance

Classes of the C++ Interface

The main programming interface to the E2953A/E2954A is based on the C++ programming language. Included with the software is also a TCL representation of the C++ interface - a shell to provide you with the capabilities of a script language. Thus All C++ calls have a TCL equivalent.

You therefore have a choice of using either the C++ interface or the TCL shell to program and configure the E2953A/E2954A InfiniBand Generator. Both interfaces can be used to integrate the E2953A/E2954A into test software and test environments.

C++ Interface

The key class within the C++ interface is the class `IGCGenerator` that contains the main methods for connecting to an InfiniBand generator. InfiniBand packets are created using the class `IGCPacket` and its derived classes. All classes and their methods are described below.

All classes can be printed using the C++ stream operator (`<<`) or the method `Print()`. Depending on the class, the output is either a textual representation of the class or a description of the class status (as with the `IGCError` class, see *"Error Class" on page 2-15*). Expressed another way, it allows you to find out what the data content of a class is.

Generator Class

The generator class `IGCGenerator` is the 'main' class needed to connect the software to a specific generator. There can be only one generator class for each physical E2953A 1x Generator for InfiniBand or E2954A 4x Exerciser for InfiniBand. However, you can handle several generator class objects simultaneously where each of these objects is connected to a different generator. A generator object can also be created with an offline connection.

The `IGCGenerator` class is derived from the class `IGCObject` which implements the `Set()` and `Get()` functions for the properties (see *"Miscellaneous Classes" on page 2-15*).

IGCGenerator Class

This is the main class for interfacing the software to the E2953A/E2954A.

For each generator that is connected, you need to have one instance of this class. The generator class also contains a dispatcher that manages the flow of received packets to registered packet handlers. For the methods relating to this class refer to *"Methods of the IGCGenerator Class" on page 2-18*.

Packet Classes

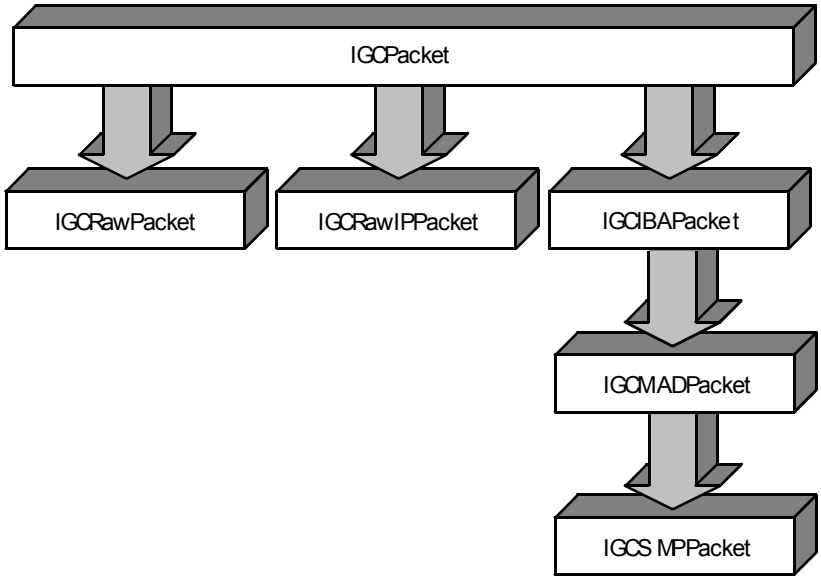
There are several packet classes (derived from the general packet class `IGCPacket`) that reflect the different basic types of packets that can be transported via an InfiniBand link. These are the raw packet, the raw packet with the IPv6 header, the InfiniBand packet and the MAD packet. The latter is derived from the InfiniBand packet. For the methods relating to this class refer to *"Methods of the IGCPacket Class" on page 2-60*.

The `IGCPacket` class itself is derived from the `IGCObject` class (see *"Miscellaneous Classes" on page 2-15*).

IGCPacket Overview

The following figure shows the derivation hierarchy of the Packet classes. You can derive additional classes from these should the need arise.

Figure 2 Hierarchy of the Packet Classes



IGCPacket Classes

The `IGCPacket` class is the base class for all classes that hold InfiniBand architecture packets (InfiniBand packets and raw packets). For the methods relating to this class, refer to *"Methods of the IGCPacket Class" on page 2-60*.

From this class the following classes are derived:

- `IGCRawPacket` class

  Raw packet with raw header. For the methods relating to this class refer to *"Methods of the IGCRawPacket Class" on page 2-69*.

- `IGCRawIPPacket` class

  Raw packet with IPv6 header. For the methods relating to this class refer to *"Methods of the IGCRawIPPacket Class" on page 2-72*.

- `IGCIBAPacket` class

  Standard InfiniBand packet. This type of packet can be either local or global. The local/global parameter applies to packets of type `IGCIBAPacket` and `IGCMADPacket` (and all packets derived from them) and determines whether a global routing header should be present in the packet or not (refer to the *InfiniBand Specification Section 5.2*). For the methods relating to this class refer to *"Methods of the IGCIBAPacket Class" on page 2-75*.

  The following class is derived from the `IGCIBAPacket` class:

  - `IGCMADPacket` class

    Special class to hold InfiniBand MAD packets. This class allows convenient access to all MAD information. For the methods relating to this class refer to *"Methods of the IGCMADPacket Class" on page 2-79*.

  The following class is derived from the `IGMADPacket` class:

  - `IGCSMPPacket` class

    Special class to hold InfiniBand SMP packets. For the methods relating to this class refer to *"Methods of the IGCSMPPacket Class" on page 2-82*.

Packet Handler Classes

The Packet Handler classes consist of the class `IGCPacketHandler` and the class `IGCPacketHandlerTcl`, where the latter is derived from the former.

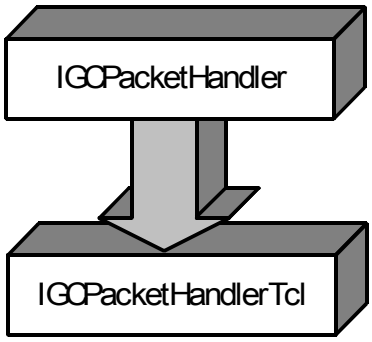
The class `IGCPacketHandlerTcl` provides the functionality for handling TCL scripts. This makes it easy for a TCL script to handle incoming packets.

There is a simple SMA (subnet management agent) implemented by a TCL script, which is also part of the software. The script also serves as an example of how to program in TCL. You can find the script under *"Installed TCL Sample Scripts" on page 1-11*.

IGCPacketHandler Overview

The following figure shows the derivation hierarchy of the Packet Handler classes.

Figure 3 Hierarchy of the Packet Handler Classes



IGCPacketHandler Class

This is an abstract base class. It basically manages two functions, both implemented as purely virtual methods that check and handle incoming packets. The methods are `CheckPacket()` and `HandlePacket()`. For the methods relating to this class refer to *"Methods of the IGCPacketHandler Class" on page 2-105*.

As a user you are free to derive additional classes from the `IGCPacketHandler` class to write your own packet handlers. Using the `IGCPacketHandlerTcl` class you can also write an entire packet handler in TCL.

The following class is derived from `IGCPacketHandler`:

- `IGCPacketHandlerTcl` class

  This class allows you to provide the generator class with TCL scripts to handle incoming packets. For the methods relating to this class refer to *"Methods of the IGCPacketHandlerTcl Class" on page 2-108*.

CallBack Classes

The class `IGCCallBack` provides the methods to handle callbacks from the API.

You cannot use the class directly, it is purely virtual and has to be derived. You need to implement method `Notify()` in your derived class. The generator uses this method to pass the callback data for handling. You are free to do whatever is necessary in this method.

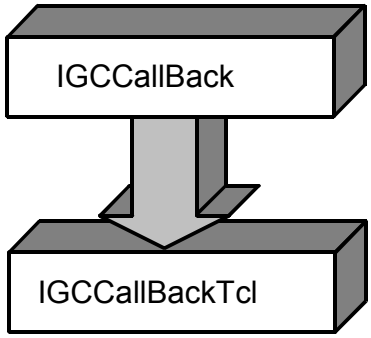
The `CallBack` classes consist of the class `IGCCallBack` and the class `IGCCallBackTcl`, where the latter is derived from the former.

The class `IGCCallBackTcl` provides the functionality for handling TCL scripts. This makes it easy for a TCL script to handle callbacks.

IGCCallBack Overview

The following figure shows the hierarchy of the `CallBack` classes.

Figure 4 Hierarchy of the CallBack Classes



IGCCallBack Class

This is an abstract base class. It basically manages two methods: `Notify` and `SetNotifyMask`. These methods are described in *"Methods of the IGCCallBack Class" on page 2-110*.

As a user you are free to derive additional classes from the `IGCCallBack` class to write your own callbacks.

IGCCallBackTcl class

This class is derived from `IGCCallBack`. It allows you to provide the generator class with TCL scripts to handle callback events. For the methods relating to this class refer to *"Methods of the IGCCallBackTcl Class" on page 2-114*.

Property Value Class

The class `IGCVal` is designed to take different kinds of property values into one type of variable. These properties control the behavior of the generator or the assembly of packets within one of the packet classes. As with all classes, the class `IGCVal` can be sent to an output stream using the C++ stream operator to get a textual representation of the property value (see *"Methods of the IGCGeneratorInfo Class" on page 2-57*).

IGCVal Class

This class can hold different data types (integers, long integers, strings, boolean and so on) up to 128 bits in length. For the methods relating to this class refer to *"Methods of the IGCVal Class" on page 2-93*.

MAD Attribute Classes

The InfiniBand Specification describes several attributes that can be carried by MAD packets. The attributes are realized as sub classes of the class `IGCMADAttribute`.

To create a MAD packet that contains one of the above attributes:

1. Create an instance of the desired attribute class.
2. Modify its properties.
3. Copy the data into the instance of a MAD Packet using the method `ToPacket()` of the class `IGCMADAttribute`.
4. Use the method `FromPacket()` from the class `IGCMADAttribute` to extract the attribute data of a MAD Packet.

"Example of Script Using MAD Attribute Classes" on page 2-9 shows an example of script using the `PortInfo` attribute.

Table 3 Correspondence between Attributes classes and InfiniBand Specification

MAD Attribute Class	InfiniBand Attribute	Section of the InfiniBand specification
IGCSubnMgmtAttribute	Subnet Management attribute	14.2 Subnet Management Class
IGCPerfMgtAttribute	Performance Management attribute	16.1 Performance Management
IGCDTAAttribute	Device Test Agent attribute	16.3 Device Management
IGCDCommMgtAttribute	Communication Management attribute	16.7 Communication Management
IGCMADAttrNotice	Notice attribute	13.4.8 Management Class Attributes
IGCMADAttrInformInfo	InformInfo attribute	13.4.8 Management Class Attributes
IGCMADAttrClassPortInfo	ClassPortInfo attribute	13.4.8 Management Class Attributes

Example of Script Using MAD Attribute Classes

The following TCL script shows how to create a MAD packet containing the `PortInfo` attribute:

```
# Creation of SMP packet with attribute PortInfo
# Create object of the attribute PortInfo
set attr
[IGCSubnMgmtAttribute_NewAttr$::IGCSubnMgmtAttribute_PortInfo]
# Create packet object
set smp [new_IGCSMPacket]
# Example: set a value of the attribute PortInfo
IGCObject_Set $attr $::IGCPortInfo_LID 0x1234
# Set attribute ID of packet to PortInfo
IGCObject_Set $smp
$::IGCMADPacket_MAD_AttributeID$::IGCSubnMgmtAttribute_PortInfo
# Copy attribute values into the smp packet
IGCMADAttribute_ToPacket $attr $smp
set id [IGCObject_Get $smp $::IGCMADPacket_MAD_AttributeID]
# Create attribute
set attr [IGCSubnMgmtAttribute_NewAttr $id]
# Copy attribute values into object
IGCMADAttribute_FromPacket $attr $smp
```

Subnet Management Attribute Classes

The Subnet Management Attributes are described by corresponding generator classes.

Table 4 Description of the Subnet Management Attributes through generator classes

Subnet Management Attribute	Class
NodeDescription	IGCNodeDescription
NodeInfo	IGCNodeInfo
SwitchInfo	IGCSwitchInfo
GUIDInfo	IGCGUIDInfo
PortInfo	IGCPortInfo
LinearForwardingTable	IGCLinearForwardingTable

All these classes are derived from the `IGCObject` class which implements the `Set()` and `Get()` methods for the properties (see *"Miscellaneous Classes" on page 2-15*).

Some of the properties for the structs are read/write, others are read only. For instance, you can copy a `NodeInfo` struct to the generator `NodeInfo` struct by using::

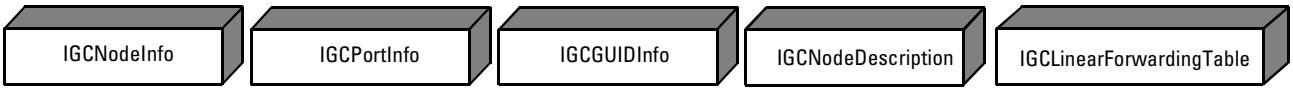
- The assignment operator

  Only the read/write properties will be copied.

- The `CopyProps()` of `IGCObject`

  In this case, you can specify to copy read-only properties.

Figure 5 Data Classes



IGCNodeInfo Class This class holds the properties for the InfiniBand `NodeInfo` struct. Each generator class has one `NodeInfo` class object. For the properties refer to *"IGCNodeInfo Properties" on page 4-6*.

IGCPortInfo Class This class holds the properties for the InfiniBand `PortInfo` struct. Each generator class has one `PortInfo` class object. For the properties refer to *"IGCPortInfo Properties" on page 4-8*.

IGCGUIDInfo Class This class holds the properties for the InfiniBand `GUIDInfo` struct. Each generator class has one `GUIDInfo` class object. For the properties refer to *"IGCGUIDInfo Properties" on page 4-7*.

IGCNodeDescription Class This class holds the properties for the InfiniBand `NodeDescription` struct. Each generator class has one `NodeDescription` class object. For the properties refer to *"IGCNodeDescription Properties" on page 4-7*.

IGCSwitchInfo Class This class holds the properties for the InfiniBand `SwitchInfo` struct. Each generator class has one `SwitchInfo` class object. For the properties refer to the InfiniBand specification.

IGCLinearForwarding Table Class This class holds the properties for the InfiniBand `LinearForwardingTable` struct. Each generator class has one `LinearForwardingTable` class object. For the properties, refer to the InfiniBand specification.

IGCPerformance Class

The class `IGCPerformance` is derived from `IGCObject` and represents a container for the result of the performance measurement that is controlled via methods in the generator class. Refer to *"Performance Measurement" on page 1-7* for more information.

The class `IGCPerformance` contains the following properties:

Table 5 Properties of the IGCPerformance Class

Property Name	Range	Default	Access	Description
RefClk	64bit	0	RW	Time in 16ns since last Measurement
RcvePyldHdr	64bit	0	RW	Received DWORDs from LRH to End of Payload, followed by EGP
RcvePyld	64bit	0	RW	Received DWORDs from Start of Payload to End of Payload, followed by EGP
RcvePkt	64bit	0	RW	Number of Received Packets followed by EGP
RcvePktBad	64bit	0	RW	Number of Received Packets followed by EBP
RcvePktLink	64bit	0	RW	Number of Received Link Packets on all VL
XmitPyldHdr	64bit	0	RW	Transmitted DWORDs from LRH to End of Payload, followed by EGP
XmitPyld	64bit	0	RW	Transmitted DWORDs from Start of Payload to End of Payload, followed by EGP
XmitPkt	64bit	0	RW	Number of Transmitted Packets followed by EGP

Property Name	Range	Default	Access	Description
XmitPktBad	64bit	0	RW	Number of Transmitted Packets followed by EBP
XmitPktLink	64bit	0	RW	Number of Transmitted Link Packets on all VL

IGCProtocolObserver Class

The protocol observer contains data related to the InfiniBand protocol. Refer to *"Link Packet and Protocol Observer" on page 1-8* for details.

The class `IGCProtocolObserver` is derived from `IGCObject` and represents a container for the result of the protocol observer that is controlled via methods in the generator class. It contains the following properties:

Table 6 Properties of the IGCProtocolObserver class

Property Name	Range	Default	Access	Description
LinkPacket_Timeout_Exceeded	16bit	0	RW	Each bit represents a VL. The Least Significant Bit is VL0. The Most Significant Bit is VL15. A '1' indicates that the timeout for the link packet was exceeded. The timeout for link packets is 65536 symbol time, with each symbol time unit being 4ns.

IGCLinkPacketStatus Class

The class `IGCLinkPacketStatus` is derived from `IGCObject` and represents a container for the result of the link packet observer that is controlled via methods in the generator class. It contains the following properties:

Table 7 Properties of the IGCLinkPacketStatus class

Property Name	Range	Default	Access	Description
All_Normal_Packets	16bit	0	RW	The Least Significant Bit represents VL0, the Most Significant Bit represents VL15. A '1' indicates that a normal link packet was received. A '0' that no normal link packet was received.
All_Init_Packets	16bit	0	RW	The Least Significant Bit represents VL0, the Most Significant Bit represents VL15. A '1' indicates that an init link packet was received. A '0' that no init link packet was received.
Received_Op	4bit	0	RW	'Op'-field in Flow Control Packet of the captured link packet.
Received_FCTBS	12bit	0	RW	'FCTBS'-field in Flow Control Packet of the captured link packet.
Received_VL	4bit	0	RW	'VL'-field in Flow Control Packet of the captured link packet.
Received_FCCL	12bit	0	RW	'FCCL'-field in Flow Control Packet of the captured link packet.
Received_LPCRC	16bit	0	RW	'LPCRC'-field in Flow Control Packet of the captured link packet.

Error Class

Error handling in the C++ as well as in the TCL interface is based on C++ exception handling. Errors are of the type `IGCError`.

IGCError Class This class handles errors and gets thrown if an error occurs. Error handling takes place via exception handling (try and catch). You are responsible for catching potential errors. For the methods relating to this class refer to *"Methods of the IGCPacketHandler Class" on page 2-105*.

Miscellaneous Classes

This group consists of the classes `IGCGeneratorList`, `IGCGeneratorInfo`, `IGCObject`, `IGCBuffer` and `IGCStatus`.

IGCGeneratorList Class This class does an automatic USB scan and creates a list of all connected generators. For the methods relating to this class refer to *"Methods of the IGCGeneratorList Class" on page 2-53*.

IGCGeneratorInfo Class This class contains a set of 'static' information. Every generator provides this information in the form of a serial number or the USB port with which it is connected (see *"Methods of the IGCGeneratorInfo Class" on page 2-57*).

IGCObject Class This class is purely virtual and is the base class for most user-accessible classes. It implements the functions `Set()` and `Get()` necessary for property control. For the methods relating to this class refer to *"Methods of the IGCObject Class" on page 2-100*.

IGCBuffer Class This class holds a buffer, for instance, for the data payload or provides space to store an unparsed packet. For the methods relating to this class refer to *"Methods of the IGCBuffer Class" on page 2-85*.

IGCStatus Class This class contains the entire status information available from one E2953A/E2954A Generator. For the methods relating to this class refer to *"Methods of the IGCStatus Class" on page 2-103*.

Methods Common to All Classes

Every class (or its base class) has the method `Print()` and supports the C++ stream operator for the user to be able to print the content of the class. The content can be either debug information or status information.

Print

Call

```
ostream & Print (ostream & o) const;
```

Description

Prints the content of the class as text representation of the specified `ostream` (in a form readable by humans). While this method is common to all classes, it is additionally mentioned in the classes `IGCGeneratorInfo`, `IGCGeneratorStatus` and `Error`. In these classes it is used to deliver the content of the internal variables of the particular class (for instance for the purpose of debugging while using TCL).

Return Value

Returns a reference to an `ostream` object with the content of the class.

Input Parameters

`o`
The stream to print into. This provides you with the possibility to print to a file or to stdout.

See also

None

Operator <<

Call `ostream & operator << (ostream & o, const <classname> & <var>);`

Description Similar to `print` but uses the C++ streaming operator.

Return Value Returns a reference to an ostream object with the content of the class.

Input Parameters

`o`
The specified ostream.

`var`
The class, the content of which is to be printed.

See also None

Methods of the IGCGenerator Class

There can only be one `IGCGenerator` class for each physical E2953A/E2954A. The generator class also holds information on the InfiniBand port `info` struct and all associated data. Since there can only be one generator per real device the copy constructor leads to an assertion.

`IN`, `OUT` and `INOUT` are markers that determine the parameter type (input or output).

Characteristic Members

The following table lists all characteristic members of the `IGCGenerator` class:

void	AssertTriggerOut (void);
void	Connect (IN ig_int32 index);
void	Disconnect (void);
void	EnableMADHandling (IN ig_bool bEnable = true);
const IGCGeneratorInfo &	GetInfo (void) const;
IGCSubnMgmtAttribute &	GetSubnMgmtAttribute (ig_int16 attr);
void	HardwareUpdate(IN ig_int32 port, IN ig_bool force = false);
void	IBLinkReset (void);
ig_bool	IsConnected(void) const;
ig_bool	IsMADHandling(void) const;
	IGCGenerator (void);
	~IGCGenerator (void);
ig_int8	LaneSkewGet(IN ig_int8 lane);
void	LaneSkewSet(IN ig_int8 lane, IN ig_int8 val);
void	LinkPacketRecRun(IN ig_int8 VL = 0x0);
void	LinkPacketRecStop();

```
void            LinkPacketStatusRead(OUT IGCLinkPacketStatus &status);
void            LinkStateWrite ( IN ig_int8 linkstate );
void            LinkTrainingStateWrite ( IN ig_int8 linkstate );
ig_int16        OperationalVLRead (void);
void            OperationalVLWrite (IN ig_int16 allVLState);
ig_int16        PerformanceCtrMaskRead(IN ig_bool direction, IN ig_int8 ctrNum);
void            PerformanceCtrMaskWrite(IN ig_bool direction, IN ig_int8  ctrNum, IN
                ig_int16 ctrMask);
void            PerformanceRead(OUT IGCPerformance &performance, IN ig_int8 ctrNum);
void            PerformanceStart();
void            PerformanceStop();
void            PacketInit ( IN IGCPacket & packet );
void            PacketSend ( IGCPacket & packet );
void            PatternActionWrite ( IN ig_int8 pattern, IN ig_int8 action );
void            PatternMaskWrite ( IN ig_int8 pattern, IN const IGCVal & mask );
void            PatternOffsetWrite ( IN ig_int8 pattern, IN ig_int32 offset );
void            PatternValueWrite ( IN ig_int8 pattern, IN const IGCVal & value );
void            Ping (void);
void            ProtocolObserverRead(OUT IGCProtocolObserver &status);
void            ProtocolObserverReset();
void            RegisterCallBack (IN CBTypes cbType, IN IGCCallBack & pCB, IN ig_bool
                atEnd = true);
void            UnregisterCallBack (IN CBTypes cbType, IN IGCCallBack & pCB);
void            RegisterPacketHandler ( IN IGCPacketHandler & handler, IN ig_bool at End =
                true );
void            Reset ( void );
void            ResetPacketSend ( void );
void            SkipTestRun (IN ig_int8 count = 0xFF, IN IGESkipMode mode = SKIP_SAME);
void            StatusRead ( IN IGCStatus & status ) const;
void            TransmitInit ( void );
void            TransmitProg ( void );
void            TransmitRun ( void );
void            TransmitSet ( IN const IGCPacket & packet );
void            TransmitStep ( void );
void            TransmitStop ( void );
```

```
void            UnregisterPacketHandler ( IN IGCPacketHandler & handler );
ig_int32        VLAllResourceRead (void);
void            VLAllResourceWrite (IN ig_int32 regVal);
ig_int8         VLResourceRead ( IN ig_int8 VL );
void            VLResourceWrite ( IN ig_int8 VL, IN ig_int8 resource );
ig_int8         VLStateRead ( IN ig_int8 VL );
void            VLStateWrite ( IN ig_int8 VL, IN ig_int8 state );
```

Inherited Members The following table lists the inherited members of the `IGCGenerator` class (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void            Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal          Get ( IN ig_int32 prop );
virtual void    Default ( void );
virtual void    CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <iggenerator.h>
```

AssertTriggerOut

```
void AssertTriggerOut ( void );
```

Description Manually asserts the trigger out signal.

Return Value None

Parameters None

See also None

Connect

```
void Connect ( IN ig_int32 portNum );
```

Description Connects the generator class to the physical generator at the USB port number `portNum`. This is the number returned by `GetPort()` (see *"Methods of the IGCGeneratorInfo Class" on page 2-57*). If the E2953A/E2954A to be connected is already in use by another generator class object, this call results in an error.

Return Value None

Input Parameters `portNum`
USB port number returned by `GetPort()`.

See also *Disconnect*

Disconnect

```
void Disconnect ( void );
```

Description Disconnects the class from the physical generator. Without an active connection all calls directly accessing the generator result in an error.

Return Value None

Input Parameters None

See also `Connect` above

EnableMADHandling

```
void EnableMADHandling ( IN ig_bool bEnable = true );
```

Description This call enables or disables MAD handling. Internally, the call registers an MAD packet handler and starts it.

Return Value None

Input Parameters `bEnable`
Enables or disables MAD handling. Takes the values `true` or `false`.

See also None

GetInfo

```
const IGCGeneratorInfo & GetInfo ( void ) const;
```

Description Returns a reference to an `IGCGeneratorInfo` object which contains information about the generator itself (USB port number, serial number, revision number, and so on).

Return Value A reference to the appropriate `IGCGeneratorInfo` object.

Input Parameters None

See also None

GetSubnMgmtAttribute

```
IGCSubnMgmtAttribute & GetSubnMgmtAttribute (ig_int16 attr);
```

Description Get subnet management attribute of the generator

Return Value A reference to the appropriate `IGCSubnMgmtAttribute` object.

Parameters `attr`
Attribute ID:

- `NodeDescription`
- `NodeInfo`
- `SwitchInfo`
- `GUIDInfo`
- `PortInfo`
- `LinearForwardingTable`

See also *"MAD Attribute Classes" on page 2-7* and *"Example of Script Using MAD Attribute Classes" on page 2-9*

HardwareUpdate

```
void HardwareUpdate(IN ig_int32 port, IN ig_bool force = false);
```

Description Updates the hardware. The versions of the firmware and FPGA on the exerciser are compared to the versions required by the current IGAPI and updated if needed. If the files containing the data of the firmware and FPGA are not found in the current directory, `HardwareUpdate` searches in the path `<InstallDir>\HW`.

Return Value None

Input Parameters

`port`
Current USB port

`force`
Force update, even if versions match

See also None

IBLinkReset

```
void IBLinkReset ( void );
```

Description Resets the link and initializes new training sequences and new link training. The settings and property values of the generator itself are not reset nor changed.

Return Value None

Input Parameters None

See also None

IsMADHandling

Call `ig_bool IsMADHandling(void) const;`

Description Returns whether MAD handling is running.

Return Value 1 if handling is running,.
0 if not.

Input Parameters None

See also None

IsConnected

Call `ig_bool IsConnected(void) const;`

Description Checks whether the generator is connected to a physical generator at the USB port.

Return Value 0: Offline
1: A generator is connected

Input Parameters None

See also *"Connect" on page 2-21*

IGCGenerator, Constructor

Call `IGCGenerator ( void );`

Description Constructor.

Without a call to the method `Connect()` the generator class is not able to process any direct accesses to hardware. By default, the generator is started in offline mode.

Return Value None

Input Parameters None

See also *IGCGenerator* Destructor

~IGCGenerator, Destructor

Call `~IGCGenerator ( void );`

Description Destructor.

If a connection is active, it is closed automatically.

Return Value None

Input Parameters None

See also *IGCGenerator* Constructor

LaneSkewGet

Call `ig_int8 LaneSkewGet(IN ig_int8 lane);`

Description Gets the skew of a lane of the transmitting side.

NOTE This method is not available for the E2953A.

Return Value Lane skew in *Symbol Times* (4ns)

Input Parameters `Lane`

The values taken by `lane` correspond to:

- `LANE_A`
- `LANE_B`
- `LANE_C`
- `LANE_D`

See also *LaneSkewSet*
LaneSkew property of the class *IGCStatus*

LaneSkewSet

Call `void LaneSkewSet(IN ig_int8 lane, IN ig_int8 val);`

Description Sets the skew of a lane for the transmitter. By setting the delays for the four lanes to different values a lane to lane skew can be generated. This register can be written anytime even during link up state but the link is likely to go down afterwards and a retraining is typically performed.

NOTE This method is not available for the E2953A.

Return Value None

Input Parameters `lane:`

The values taken by `lane` correspond to:

- `LANE_A`
- `LANE_B`
- `LANE_C`
- `LANE_D`

`val`

Lane skew in Symbol Times (4ns for IBx1 and IBx4). The minimum value is 0, the maximum value is 9.

See also *LaneSkewGet*
LaneSkew property of the class *IGCStatus*

LinkPacketRecRun

Call `void LinkPacketRecRun(IN ig_int8 VL = 0x0);`

Description Starts monitoring link packets received by the generator. All virtual lanes are monitored for occurring `Normal Link Packets` and `Init Link Packets`. For the virtual lane set in the parameter `VL` the first occurring link packet will be captured.

Return Value None

Input Parameters `VL`
The first link packet on the virual lane will be captured.

See also *LinkPacketRecStop*, *LinkPacketStatusRead*

LinkPacketRecStop

Call `void LinkPacketRecStop();`

Description Stops monitoring link packets received by the generator.

Return Value None

Input Parameters None

See also *LinkPacketRecRun*, *LinkPacketStatusRead*

LinkStateWrite

Call `void LinkStateWrite ( IN ig_int8 linkstate );`

Description Sets the link state machine into one of the states listed in `Input Parameters`. This is normally performed by the Subnet Management software but if this is not present in the InfiniBand network you may need to do it manually.

Return Value None

Input Parameters `linkstate`
The values taken by `linkstate` are:

- `LINKCMD_DOWN`
  The link state machine is down.
- `LINKCMD_ARM`
  The link state machine is armed.
- `LINKCMD_ACTIVE`
  The link state machine is active.

See also None

LinkPacketStatusRead

Call `void LinkPacketStatusRead(OUT IGCLinkPacketStatus &status);`

Description Reads out the result when monitoring link packets using `LinkPacketRecRun` and `LinkPacketRecStop`.

Return Value None

Input Parameters Instance of `IGCLinkPacketStatus` where result will be stored

See also *LinkPacketRecRun*, *LinkPacketRecStop*

LinkTrainingStateWrite

Call `void LinkTrainingStateWrite ( IN ig_int8 linkstate );`

Description You have to call this function to change the state of the link training state machine from `disabled` to `sleep`. Without this call the generator cannot begin establishing an InfiniBand link.

Return Value None

Input Parameters `linkstate`
The values taken by `linkstate` are:

- `LINKTRAINCMD_DISABLED`
  The link training state machine is disabled.
- `LINKTRAINCMD_SLEEP`
  The link training state machine is inactive (sleeps).
- `LINKTRAINCMD_POLL`
  The link training state machine is polling.
- `LINKTRAINCMD_INITIATEERRORRECOVERY`
  The link training state machine is in recovery

See also *"Reset" on page 2-43*

OperationalVLRead

Call `ig_int16 OperationalVLRead (void);`

Description Reads out the virtual lanes enabled on the E2953A/E2954A.

Return Value Each bit of the return value represents the enable bit of the corresponding virtual lane. `Bit0` holds the enable bit for `VL0` and so forth. Set to 1 the virtual lane is enabled.

Input Parameters `None`

See also None

OperationalVLWrite

Call `void OperationalVLWrite (IN ig_int16 allVLState);`

Description Enables the virtual lanes on the E2953A/E2954A.

Return Value None

Input Parameters Each bit of the return value represents the enable bit of the corresponding VL. `Bit0` holds the enable bit for `VL0` and so forth. Set to 1 the virtual lane is enabled.

See also None

PacketInit

Call `void PacketInit ( IN IGCPacket & packet );`

Description This method initializes a packet with all the global properties available to the generator (source lid, source gid, and so on). If you do not use this method, you must set the packet header information by some other means. Note, that the generator is capable of sending out uninitialized packets, but such packets do not conform to the InfiniBand specification.

Return Value None

Input Parameters `packet`
The packet to be initialized.

See also None

PacketSend

Call `void PacketSend ( IGCPacket & packet );`

Description Send a packet directly out of the IB link. This method bypasses the transmit memory. Its purpose is to allow you to send high priority packets or MAD packets at any time without the need for extensive memory programming.

Return Value None

Input Parameters `packet`
The packet to be sent.

See also *"ResetPacketSend" on page 2-44*

PatternActionWrite

Call `void PatternActionWrite ( IN ig_int8 pattern, IN ig_int8 action );`

Description Determines which pattern matcher will be used to match the incoming pattern and the action that will take place when a pattern match occurs (refer to the *Agilent Technologies E2950 Series InfiniBand Exerciser User Guide* for more information about the pattern matcher).

Return Value None

Input Parameters

`pattern`
`Selects t`he pattern matcher that will be used to check for pattern 'hits'. Values taken by `pattern` can be:

- `PATTERN_GENERIC`
  The Generic pattern matcher will be used.
- `PATTERN_BUFFER`
  The MAD Buffer pattern matcher will be used.
- `PATTERN_LOWERMEMORY`
  The Low Receive Pattern Memory pattern matcher will be used.
- `PATTERN_UPPERMEMORY`
  The High Receive Pattern Memory pattern matcher will be used.

`action`
The action assigned to the selected pattern matcher. Values taken by `action` can be:

- `ACTION_NONE`
  This is defined to 0. No action follows.
- `ACTION_TRIGGEROUT`
  Asserts the external trigger-out line for all packets for which the extracted bits match the set value.

- ACTION_STEPSTROBE

  Launches a step register strobe for all packets for which the extracted bits match the set value.

- ACTION_DISCARDNOHIT

  When set to 1, discards all incoming packets for which the extracted bits do not match the set value.

  The default for packets is pass-through.

- ACTION_NEGATEPATTERN

  This negates the pattern that determines the pattern match .

You can select any combination of these actions using OR.

See also *"PatternMaskWrite"* below, *"PatternOffsetWrite" on page 2-36*and *"PatternValueWrite" on page 2-37*

PatternMaskWrite

Call

```
void PatternMaskWrite ( IN ig_int8 pattern, IN const IGCVal & mask );
```

Description Writes the pattern mask in the selected pattern matcher. Pattern matching will take place only on bits where the mask contains a 1.

Return Value None

Input Parameters

`pattern`
`Selects t`he pattern matcher where the mask is to be written. Values taken by `pattern` can be:

- PATTERN_GENERIC
  The mask in the Generic pattern matcher is written.
- PATTERN_BUFFER
  The mask in the MAD Buffer pattern matcher is written.
- PATTERN_LOWERMEMORY
  The Low Receive Pattern Memory pattern matcher is written.

  PATTERN_UPPERMEMORY
  The mask in the High Receive Pattern Memory pattern matcher is written.

`mask`
The pattern will be checked only on the bits where the mask contains a 1.

See also *"PatternActionWrite" on page 2-34*, *"PatternOffsetWrite"* below and *"PatternValueWrite" on page 2-37*.

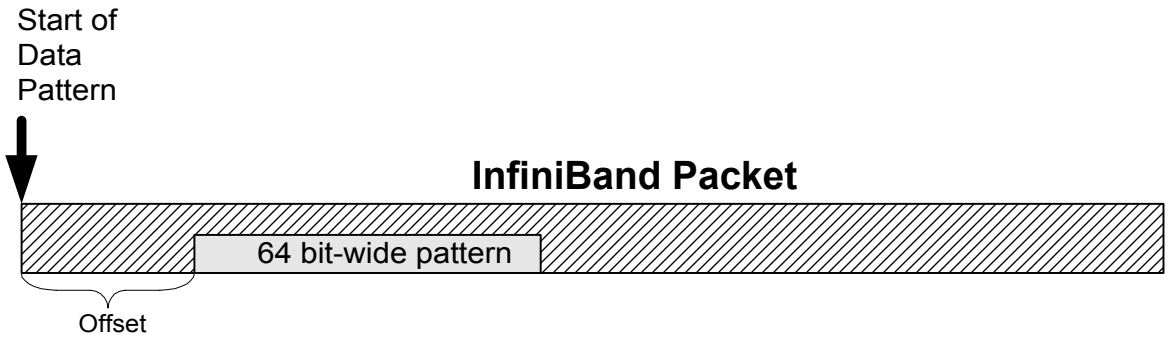
PatternOffsetWrite

Call

```
void PatternOffsetWrite ( IN ig_int8 pattern, IN ig_int32 offset );
```

Description Defines the offset within a packet where the pattern is applied. The 64 bit wide pattern compares with the content of the data packet at the position determined by 'offset' as shown in the following figure:

Figure 6 Defining the Offset for Pattern Comparison



Return Value None

Input Parameters `pattern`
`Selects t`he pattern matcher for which the offset is to be defined. Values taken by `pattern` can be:

- `PATTERN_GENERIC`
  The offset for the Generic pattern matcher is to be defined.
- `PATTERN_BUFFER`
  The offset for the MAD Buffer pattern matcher is to be defined.
- `PATTERN_LOWERMEMORY`
  The offset for the Low Receive Pattern Memory pattern matcher is to be defined.

- PATTERN_UPPERMEMORY
  The offset for the High Receive Pattern Memory pattern matcher is to be defined.

offset
The offset value has to be DWORD aligned.

See also *"PatternActionWrite" on page 2-34*, *"PatternMaskWrite" on page 2-35* and *"PatternValueWrite"* below.

PatternValueWrite

Call
```
void PatternValueWrite ( IN ig_int8 pattern, IN const IGCVal & value );
```

Description Writes a 64-bit comparison pattern into the value register of the selected pattern matcher. The incoming patterns will be compared against this pattern value, but only on those bits where the mask has been set to 1 (refer to *"PatternMaskWrite" on page 2-35*).

Return Value None

Input Parameters

pattern
Selects the pattern matcher where the pattern value is to be written. Values taken by pattern can be:

- PATTERN_GENERIC
  The pattern value will be written into the Generic pattern matcher.
- PATTERN_BUFFER
  The pattern value will be written into the MAD Buffer pattern matcher.
- PATTERN_LOWERMEMORY
  The pattern value will be written into the Low Receive Pattern Memory pattern matcher.
- PATTERN_UPPERMEMORY
  The pattern value will be written into the High Receive Pattern Memory pattern matcher.

`value`
A 64-bit value to compare against. The value should "match" the binary mask as set by `PatternMaskWrite`. Remember, the pattern matchers will perform a comparison only on the bits where the mask has been set to 1.

See also *"PatternActionWrite" on page 2-34*, *"PatternMaskWrite" on page 2-35* and *"PatternOffsetWrite" on page 2-36*

PerformanceCtrMaskRead

Call

```
ig_int16 PerformanceCtrMaskRead(IN ig_bool direction,
                                IN ig_int8 ctrNum);
```

Description Reads out the virtual lanes monitored by a performance counter. The performance measurement must not be running when this command is executed.

Return Value The mask is returned. The Least Significant Bit represents VL0 and the Most Significant Bit represents VL15. If a bit is set to '1', the packets on the virtual lane are included in the count.

Input Parameters

`direction`
Indicates if this is the mask for virtual lanes for the outgoing (`Xmit`)or incoming packets (`Rcve`).

`ctrNum`
Counter number (`PerfCtr1` or `PerfCtr2`)

See also *PerformanceCtrMaskWrite*

PerformanceCtrMaskWrite

Call

```
void PerformanceCtrMaskWrite(IN ig_bool direction,
                             IN ig_int8  ctrNum,
                             IN ig_int16 ctrMask);
```

Description Writes the mask corresponding to the virtual lanes monitored by a performance counter. The performance measurement must not be running when this command is executed.

Return Value None

Input Parameters

`direction`

Indicates if this is the mask for VLs for the outgoing (`Xmit`) or incoming (`Rcve`) packets.

`ctrNum`

Counter number (`PerfCtr1` or `PerfCtr2`)

`ctrMask`

The Least Significant Bit represents VL0 and the Most Significant Bit represents VL15. If a bit is set to 1, the packets on the virtual lane are included in the count.

See also *PerformanceCtrMaskRead*

PerformanceRead

Call
```
void PerformanceRead(OUT IGCPerformance &performance,
                     IN ig_int8 ctrNum);
```

Description Reads out the values of the performance measurement.

Return Value None

Input Parameters

`performance`
`Reference to IGCPerformance` object, into which the result will be copied.

`ctrNum`
Counter number of the desired counter

See also *PerformanceStart*, *PerformanceStop*

PerformanceStart

Call
```
void PerformanceStart();
```

Description Starts the performance measurement.

Return Value None

Input Parameters None

See also *PerformanceStop*

PerformanceStop

Call
```
void PerformanceStop();
```

Description Stops the performance measurement.

Return Value None

Input Parameters None

See also *PerformanceStart*

Ping

Call `void Ping (void);`

Description The Error LED on the connected generator starts to flash.

Return Value None

Input Parameters None

See also None

ProtocolObserverRead

Call `void ProtocolObserverRead(OUT IGCProtocolObserver &status);`

Description Reads out the protocol observer status from the generator.

Return Value None

Input Parameters `status`
`Reference to IGCProtocolObserver` object into which the result will be copied.

See also *ProtocolObserverReset*

ProtocolObserverReset

Call `void ProtocolObserverReset();`

Description Resets the protocol observer values

Return Value None

Input Parameters None

See also *ProtocolObserverReset*

RegisterCallBack

Call

```
void RegisterCallBack (IN CBTypes cbType,
                       IN IGCCallBack & pCB,
                       IN ig_bool atEnd = true);
```

Description Registers a call back.

Return Value None

Input Parameters

`cbType`

Call back Type corresponding to:

- `CB_STATUS`
- `CB_PROGRESS`
- `CB_PACKETSEND`

`pCB`

Reference to `CallBack` object. It must be derived from the class `IGCCallBack`.

`atEnd`

Position in callback queue:

- `true` : the call back will be positioned at the end of the call back queue
- `false`: the call back will be positioned at the beginning to the call back queue

See also *"UnregisterCallBack" on page 2-48*

RegisterPacketHandler

Call `void RegisterPacketHandler ( IN IGCPacketHandler & handler );`

Description Registers a packet handler with the generator. All registered packet handlers are served on the "first come (first registered) – first served" basis.

Return Value None

Input Parameters `handler`
The packet handler that is to be registered.

`atEnd`

Position the handler is inserted in the handler queue.

See also *"UnregisterPacketHandler" on page 2-49*

Reset

Call `void Reset ( void );`

Description Resets the generator. In order to establish the InfiniBand link again, you have to call either `IBLinkReset()` or `LinkTrainingStateWrite(LINKTRAINCMD_POLL)`.

`Reset` does not delete the contents of the transmit memory.

Return Value None

Parameters None

See also *"IBLinkReset" on page 2-24* and *"LinkTrainingStateWrite" on page 2-31*

ResetPacketSend

Call `void ResetPacketSend ( void );`

Description Deletes the send buffer. You can use this method when a packet cannot be sent out (the link is down or the cable is disconnected) and you wish to discard the packet rather than waiting that it is sent out.

Return Value None

Input Parameters None

See also *"PacketSend" on page 2-33*

SkipTestRun

Call
```
void SkipTestRun (IN ig_int8 count = 0xFF, IN IGESkipMode mode =
SKIP_SAME);
```

Description As specified in the InfiniBand specification, the E2953A/E2954A inserts by default skip ordered sets (i.e. the sequence of a COMMA symbol followed by three SKIP symbols) that are used by the DUT to calculate the lane skew.
In addition, you can insert a special SKIP ordered set that simulates a repeater using the `SkipTestRun` method. In this ordered set, SKIP symbols are omitted and replaced by IDLE symbols.

Return Value None

Input Parameters

`count`
Number of special skip insertions

`mode`
`mode` can have following values:

- `SKIP_SAME` (keep previous set value)
- `SKIP_1` (DWORD that is inserted looks like this `IDLE COM SKP SKP`)
- `SKIP_2` (DWORD that is inserted looks like this `IDLE IDLE COM SKP`)
- `SKIP_3` (DWORD that is inserted looks like this `COM SKP SKP SKP`)

StatusRead

Call `void StatusRead ( IN IGCStatus & status ) const;`

Description Supplies the specified `IGCStatus` class with the latest status information. The status is a snapshot of the current hardware state.

Return Value None

Input Parameters `status`
Hardware status information. The input is a reference to an object of type `status` that you need to create. The generator then fills the status object with all the relevant data. For detailed information refer to *"Status Properties" on page 4-3*.

See also None

TransmitInit

Call `void TransmitInit ( void );`

Description Initializes the transmit memory programming array. The call has to be made before filling the memory.

Return Value None

Parameters None

See also None

TransmitProg

Call `void TransmitProg ( void );`

Description Programs the generator memory with the data contained in the transmit memory buffer.

Return Value None

Parameters None

See also None

TransmitRun

Call `void TransmitRun ( void );`

Description Starts sending packets out of the transmit memory.

Return Value None

Parameters None

See also None

TransmitSet

Call `void TransmitSet ( IN const IGCPacket & packet );`

Description Programs a packet into the transmit memory buffer.

Return Value None

Input Parameters `packet`
Contents of the packet to be programmed into the transmit memory buffer.

See also None

TransmitStep

Call `void TransmitStep ( void );`

Description Functions as *continue* if you have selected a certain packet to wait for a software strobe. It causes a packet that is currently waiting in line in the transmit memory to be sent out onto the InfiniBand link. The transmit memory then runs until it reaches the next packet that has been set on hold with the *wait for step* property.

If you have set up a lot of packets in the transmit memory which you intend to send out in response to a received packet of some kind or in response to some other software controlled event, you can specify in the packet properties (see *"Packet Properties" on page 4-11*) that a certain packet should wait for a step signal (`IGP_WaitStep`). This can be invoked either by one of the pattern terms or by a manual software call to `TransmitStep()`.

Return Value None

Parameters None

See also None

TransmitStop

Call `void TransmitStop ( void );`

Description Stops sending out packets from the transmit memory.

Return Value None

Parameters None

See also None

UnregisterCallBack

Call
```
void UnregisterCallBack (IN CBTypes cbType,
                         IN IGCCallBack & pCB);
```

Description Unregisters a callback

Return Value None

Input Parameters `cbType`

Type of callback:

- `CB_STATUS`
- `CB_PROGRESS`
- `CB_PACKETSEND`

`pCB`

`Callback` class. It must be derived from the class `IGCCallBack`.

See also None

UnregisterPacketHandler

Call `void UnregisterPacketHandler ( IN IGCPacketHandler & handler );`

Description Deletes a packet handler registration. Note that if the object `IGCPacketHandler` gets deleted or gets out of scope, it is automatically unregistered (the software calls the destructor of the object).

Return Value None

Input Parameters `handler`
The packet handler to be unregistered.

See also *"RegisterPacketHandler" on page 2-43*

VLAllResourceRead

Call `ig_int32 VLAllResourceRead (void);`

Description Reads out in which resource packets received on the VLs will be stored.

Return Value This register holds the two-bit resource information per virtual lane:

- 00 → Infinite sink
- 01 → Receive Buffer
- 10 → Lower Memory
- 11 → Upper Memory

The bits 1 and 0 hold the resource information for VL0 and the bits 3 and 2 for VL1 and so forth.

Input Parameters None

See also *VLAllResourceWrite*

VLAllResourceWrite

Call `void VLAllResourceWrite (IN ig_int32 regVal);`

Description Writes in which part of the receive memory (resource) packets received on the VLs will be stored.

Return Values None

Input Parameters `regVal`
This register holds the two bit resource information per virtual lane.

- 00 → Generic (Infinite sink)
- 01 → Receive Buffer
- 10 → Lower Memory
- 11 → Upper Memory

The bits 1 and 0 hold the resource information for VL0 and the bits 3 and 2 for VL1 and so forth.

See also *VLAllResourceRead*

VLResourceRead

Call `ig_int8 VLResourceRead ( IN ig_int8 VL );`

Description Reads back the resource for the specified VL.

Return Value The read resource for the virtual lane. For resource values, refer to `VLResourceWrite` below.

Input Parameters `VL`
The virtual lane to be interrogated.

See also *VLResourceWrite* below

VLResourceWrite

Call `void VLResourceWrite ( IN ig_int8 VL, IN ig_int8 resource );`

Description Defines the resource for each of the enabled VLs.

Return Value None

Input Parameters `VL`
The virtual lane for which a resource is to be assigned.

`resource`
The following resources can be assigned:

- `VLRES_BUFFER`
  Assigns the receive buffer
- `VLRES_UPPERMEMORY`
  Assigns the upper half of the receive memory
- `VLRES_LOWERMEMORY`
  Assigns the lower half of the receive memory
- `VLRES_DISCARD`
  Enables a virtual lane but discards all incoming packets. This causes the E2953A/E2954A to act as data sink with unlimited receive memory.

See also `VLResourceRead` above

VLStateRead

Call `ig_int8 VLStateRead ( IN ig_int8 VL );`

Description Reads the current state of the virtual lane (VL). Responses are 0 for disabled and 1 for enabled.

Return Value Current state of the virtual lane. For states and return values refer to `VLStateWrite` below.

Input Parameters `VL`
The virtual lane to be interrogated.

See also *VLStateWrite* below

VLStateWrite

Call `void VLStateWrite ( IN ig_int8 VL, IN ig_int8 state );`

Description Writes the state of the virtual lane into the hardware.

Return Value None

Input Parameters `VL`
The virtual lane to be set.

`state`

- `VLSTATE_OFF`
  The virtual lane is disabled.
- `VLSTATE_ON`
  The virtual lane is enabled. Disabling a virtual lane means that no credits are given out for this virtual lane.

See also *VLStateRead* above

Methods of the IGCGeneratorList Class

This class creates a list of all connected generators. When the class is created it scans the entire USB bus for all connected E2953A/E2954A InfiniBand generators.

The following table lists all characteristic members of the `IGCGeneratorList` class:

	`IGCGeneratorList ( void );`
	`~IGCGeneratorList ( void );`
`int`	`Count ( void ) const;`
`const`	`IGCGeneratorInfo & Get ( IN int index ) const;`
`const`	`IGCGeneratorInfo & operator[] ( IN int index ) const;`
`void`	`Rescan ( void );`

Include Files `#include <iggeneratorlist.h>`

IGCGeneratorList, Constructor

Call `IGCGeneratorList ( void );`

Description Constructor. It scans the USB bus for all connected devices.

Return Value None

Parameters None

~IGCGeneratorList, Destructor

Call `~IGCGeneratorList ( void );`

Description Destructor of the class.

Return Value None

Parameters None

See also None

Count

Call `int Count ( void ) const;`

Description Returns the number of generators found.

Return Value The number of generators found.

Parameters None

See also None

Get

Call `const IGCGeneratorInfo & Get ( IN int index ) const;`

Description Returns the `IGCGeneratorInfo` class with the index `index`. This call can be used from TCL.

Return Value Reference to the `IGCGeneratorInfo` object.

Input Parameters `index`
Generator index.

See also None

Operator[]

Call `const IGCGeneratorInfo & operator[] ( IN int index ) const;`

Description `Operator[]` provides access to the `IGCGeneratorInfo` class that contains serial number and port information for the generator. This type of operator cannot be translated into TCL. If working with TCL use the `Get()` function above.

Return Value Reference to the `IGCGeneratorInfo` object.

Input Parameters `index`
Generator index.

See also None

Rescan

Call `void Rescan ( void );`

Description Rescans the USB bus.

Return Value None

Parameters None

See also None

Methods of the IGCGeneratorInfo Class

This class provides 'static' information needed from a generator. You can use this information to connect a generator either via a USB port or by using its serial number.

The following table lists all characteristic members of the `IGCGeneratorInfo class`:

```
int         GetPort ( void ) const;
const       char * GetSerial ( void ) const;
const       char * GetProductString ( void ) const;
ostream     & Print (ostream & o) const;
```

Include Files `#include <iggeneratorinfo.h>`

GetPort

Call `int GetPort ( void ) const;`

Description Returns the USB port number.

Return Value USB port number.

Parameters None

See also *"Connect" on page 2-21*

GetSerial

Call `const char * GetSerial ( void ) const;`

Description Returns the serial number of the generator.

Return Value Pointer to the serial number.

Parameters None

See also None

GetProductString

Call `const char * GetProductString ( void ) const;`

Description Returns the product string of the generator (either “E2953A Generator InfiniBand by 1” or “E2954A Generator InfiniBand by 4”)

Return Value Pointer to the product string.

Parameters None

See also None

Print

Call `ostream & Print (ostream & o) const;`

Description Prints a (human readable) list of the connected generators to the specified stream.

Return Value Returns a reference to an ostream object with a list of connected generators.

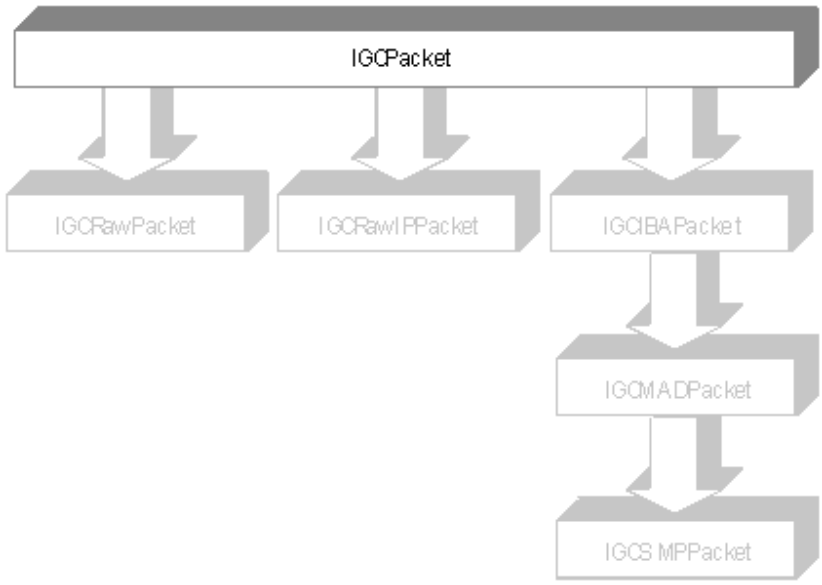
Input Parameters o
The stream to print into. This provides you with the possibility to print to a file or to stdout.

See also None

Methods of the IGCPacket Class

The class `IGCPacket` is not for direct use by the programmer. The classes `IGCIBAPacket`, `IGCRawIPPacket`, `IGCRawPacket`, `IGCMADPacket` and IGCSMPPacket are derived from this class and provide the user interface to programming the packets. The constructor is protected. Use constructors from derived classes for construction and `DeletePacket()` for deletion. (While the use of destructors from derived classes is possible, it is recommended to use `DeletePacket()` instead.)

Figure 7 IGCPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCPacket` class:

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket *         Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_int32    GetICRC (void) const;
```

```
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int32            GetType ( void ) const;
ig_int16            GetVCRC (void) const;
virtual ig_bool     HasPayload (void) const;
virtual             ~IGCPacket ();
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
```

Inherited Members The following table lists the inherited members of the `IGCPacket` class (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void            Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal          Get ( IN ig_int32 prop );
virtual void    Default ( void );
virtual void    CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <igpacket.h>
```

AppendBuffer

Call

```
void AppendBuffer ( OUT IGCBuffer & buffer ) const;
```

Description Appends a packet to a byte stream buffer. Several packets can be packed into a buffer to form a sequence.

Return Value None

Output Parameters

`buffer`
Reference to `IGCBuffer` object. A buffer containing the appended packet.

See also None

AppendPayloadBuffer

Call `void AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;`

Description Appends a packet's payload to a byte stream buffer. The payload of several packets can be packed together this way (for instance for the purpose of recombining a message).

Return Value None

Output Parameters `buffer`
Reference to `IGCBuffer` object. A buffer containing the appended payload.

See also None

Clone

Call `IGCPacket * Clone ( void ) const;`

Description Clones a packet. Returns pointer to new packet of the same type. Replaces the "virtual copy constructor", which is not available in C++.

Return Value `IGCPacket` object
Pointer to the new packet.

Parameters None

See also None

DeletePacket

Call `void DeletePacket ( void );`

Description Calls the destructor.

Return Value None

Parameters None

See also *"IGCPacket, Destructor" on page 2-66*

GetActualLength

Call `virtual ig_int16 GetActualLength ( void ) const;`

Description Returns actual packet length.
The call `SetPacketLenght ( GetActualLength() );` automatically sets the correct packet length.

Return Value A 16-bit integer holding the actual packet length.

Parameters None

See also *"SetPacketLength" on page 2-67*

GetICRC

Call `virtual ig_int32 GetICRC (void) const;`

Description Returns the ICRC of the packet

Return Value The ICRC is returned

Parameters None

GetType

Call `ig_int32 GetType ( void ) const;`

Description Returns the type of packet. Five predefined packet types can be returned. You can add your own definitions as required.

Return Value The packet type. Valid values are:
`PACKET_UNDEFINED`
`PACKET_IBATRANSPORT`
`PACKET_RAW`
`PACKET_RAWIPV6`
`PACKET_IBAMAD`
`PACKET_IBASMP`

Parameters None

See also None

GetPayload

Call `virtual void GetPayload (OUT IGCBuffer & bufPld) const;`

Description Returns the payload of packet.

Return Value None

Parameters `bufPld`
Buffer in which payload data will be stored

See also SetPayload

GetVCRC

Call `ig_int16 GetVCRC (void) const;`

Description Returns the VCRC of the packet.

Return Value `The VCRC`

Parameters None

See also None

HasPayload

Call `virtual ig_bool HasPayload (void) const {return IGD_TRUE;}`

Description Returns if this type of packet has a payload

Return Value The packet type. Valid values are:
`IGD_TRUE`
`IGD_FALSE`

Parameters None

See also None

IGCPacket, Destructor

Call `virtual ~IGCPacket ();`

Description Destructor.

Return Value None

Parameters None

See also *"DeletePacket" on page 2-63*

NewPacket

Call `IGCPacket * NewPacket ( IN & IGCBuffer databuffer );`

Description Takes the byte stream buffer `databuffer` and creates a new packet out of it. This function is static and can be used without the need of a packet object.

Return Value `IGCPacket` object for the new packet.

Input Parameters `databuffer`
The byte stream buffer containing data for the new packet.

See also None

SetPacketLength

Call `void SetPacketLength ( IN ig_int16 length );`

Description Sets the packet length within the local routing header to the specified length. You can create falsified packets by specifying an incorrect length.

Return Value None

Input Parameters `length`
A 16-bit integer specifying the packet length in bytes (see *"Local Routing Header Properties" on page 4-14*). The length must be a multiple of four bytes. The packet length is usually calculated automatically using `SetPayload`, `SetPRBSPayloadSize` (depending on the value of the property `IGP_UsePRBS`), the call `Init()` and the value of the property `IGP_UsePRBS` (refer to *"Generic Packet Properties" on page 4-12*).

See also *"GetActualLength" on page 2-63*

SetPayload

Call `void SetPayload ( IN const & IGCBuffer dataarray );`

Description Sets the payload to the data array. The length of the payload buffer must be a multiple of four bytes. You must add zero to three padding bytes and set `BTH_PadCnt` accordingly.

Return Value None

Input Parameters `dataarray`
Reference to the `IGCBuffer` object. The `IGCBuffer` class holds a buffer (`dataarray`) for a data payload.

See also *GetPayload*

SetPRBSPayloadSize

Call `virtual void SetPRBSPayloadSize (IN ig_size size);`

Description Sets the size of the PRBS payload.

Return Value None

Input Parameters `Size`
Payload size in bytes. The size must be a multiple of 4 bytes.

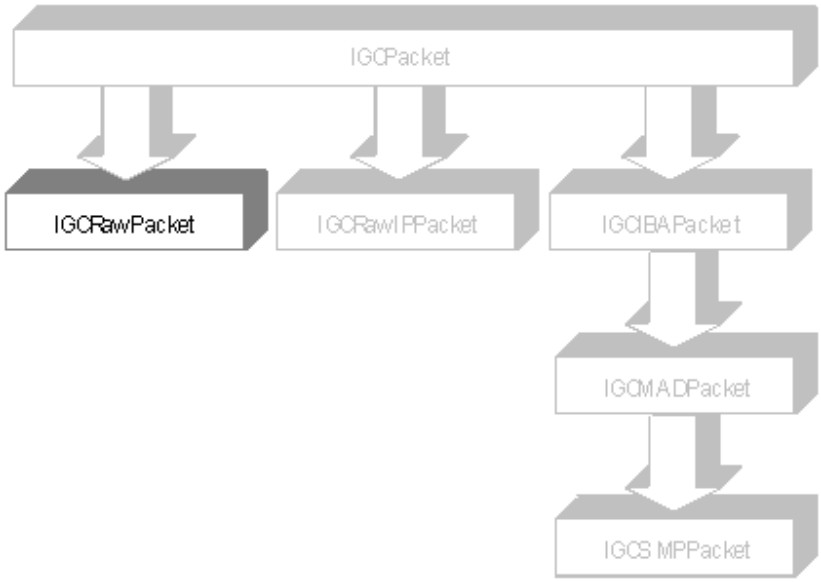
See also None

Methods of the IGCRawPacket Class

The class `IGCRawPacket` holds an InfiniBand raw packet. Only the local routing header is present in this type of packet.

Note that the use of `DeletePacket()` is the recommended method of calling a destructor even though this class has its own destructor. `DeletePacket()` is defined in the base class `IGCPacket` (see *"DeletePacket" on page 2-63*).

Figure 8 IGCRawPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCRawPacket` class:

```
                IGCRawPacket ( void );
  virtual       ~IGCRawPacket ( void );
```

Inherited Members The following table lists the inherited members of the `IGCPacket` class that are recommended for direct use (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket           Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_bool     HasPayload (void) const;
ig_int32            GetType ( void ) const;
virtual ig_int32    GetICRC (void) const;
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int16            GetVCRC (void) const;
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
void                Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal              Get ( IN ig_int32 prop );
virtual void        Default ( void );
virtual void        CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <igrawpacket.h>
```

IGCRawPacket, Constructor

Call `IGCRawPacket ( void );`

Description Constructor.

Return Value None

Parameters None

See also None

IGCRawPacket, Destructor

Call `virtual ~IGCRawPacket ( void );`

Description Destructor.

Return Value None

Parameters None

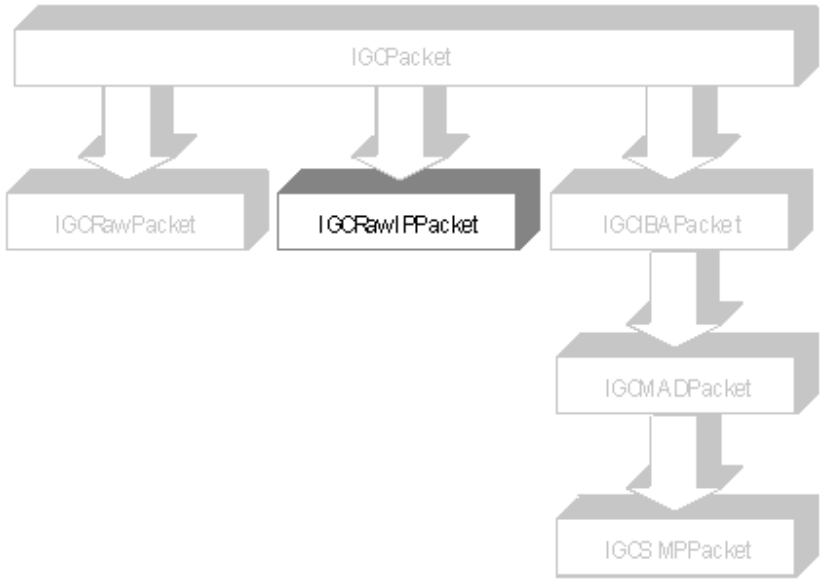
See also None

Methods of the IGCRawIPPacket Class

This class holds raw IPv6 packets that pass through the InfiniBand network.

Note that the use of `DeletePacket()` is the recommended method of calling a destructor even though this class has its own destructor. `DeletePacket()` is defined in the base class `IGCPacket` (see *"DeletePacket" on page 2-63*).

Figure 9 IGCRawIPPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCRawIPPacket` class:

```
                IGCRawIPPacket ();
virtual         ~IGCRawIPPacket ( void );
```

Inherited Members The following tables list all inherited members of the `IGCPacket` class that are recommended for direct use (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket *         Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_int32    GetICRC (void) const;
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int32            GetType ( void ) const;
ig_int16            GetVCRC (void) const;
virtual ig_bool     HasPayload (void) const;
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
void                Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal              Get ( IN ig_int32 prop );
virtual void        Default ( void );
virtual void        CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files `#include < igippacket.h>`

IGCRawIPPacket, Constructor

Call `IGCRawIPPacket ();`

Description Constructor.

Return Value None

Parameters None

See also None

IGCRawIPPacket, Destructor

Call `virtual ~IGCRawIPPacket ( void );`

Description Destructor.

Return Value None

Parameters None

See also None

Methods of the IGCIBAPacket Class

The class `IGCIBAPacket` is derived directly from the `IGCPacket` class. It holds a standard InfiniBand packet. The functions `Set()` and `Get()` necessary to manipulate packet properties are derived from the base class `IGCObject`. (This is true for all packet classes.)

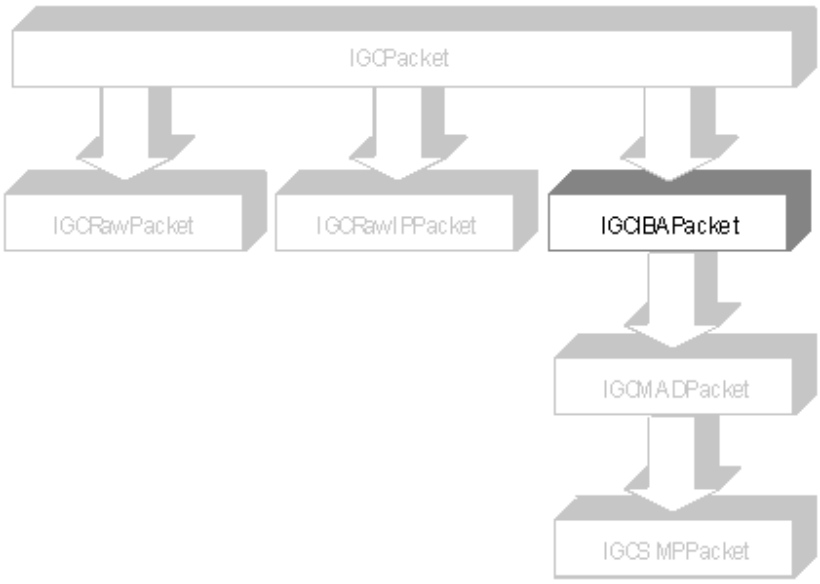
Note that the use of `DeletePacket()` is the recommended method of calling a destructor even though this class has its own destructor. `DeletePacket()` is defined in the base class `IGCPacket` (see *"DeletePacket" on page 2-63*).

The constructor is available in two versions:

- as a default constructor which requires an `Init()` call for initialization
- as a constructor for the class with `Opcode` and `IsGlobal` parameter.

Refer also to *"Sending Packets" on page 1-2*.

Figure 10 IGCIBAPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCIBAPacket` class that are recommended for direct use:

```
void            Init (IN Opcode code, IN ig_bool IsGlobal);
                IGCIBAPacket (IN Opcode code, IN ig_bool IsGlobal);
                IGCIBAPacket ( void );
virtual         ~IGCIBAPacket ( void );
```

Inherited Members

The following table lists the inherited members of the `IGCPacket` class (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket *         Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_int32    GetICRC (void) const;
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int32            GetType ( void ) const;
ig_int16            GetVCRC (void) const;
virtual ig_bool     HasPayload (void) const
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
void                Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal              Get ( IN ig_int32 prop );
virtual void        Default ( void );
virtual void        CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <igibapacket.h>
```

Init

Call `void Init ( IN Opcode code, IN ig_bool IsGlobal );`

Description Initializes the InfiniBand packet.

Return Value None

Input Parameters `code`
For opcodes refer to *"Enumeration Definitions" on page 3-1*.

`IsGlobal`
Boolean value that determines whether the packet carries a global routing header.

See also None

IGCIBAPacket, Default Constructor

Call `IGCIBAPacket ( void );`

Description Default constructor. You have to call the `Init()` method before using the packet.

Return Value None

Parameters None

See also *IGCIBAPacket*, Constructor for the Class and *IGCIBAPacket* Destructor below

IGCIBAPacket, Constructor for the Class

Call `IGCIBAPacket (IN Opcode code, IN ig_bool IsGlobal);`

Description Constructor for the class.

Return Value None

Input Parameters `code`
For opcodes refer to *"Enumeration Definitions" on page 3-1*

`IsGlobal`
Boolean value that determines whether the packet carries a global routing header.

See also *IGCIBAPacket*, Default Constructor above and *IGCIBAPacket* Destructor below

~IGCIBAPacket, Destructor

Call `virtual ~IGCIBAPacket( void );`

Description Destructor.

Return Value None

Parameters None

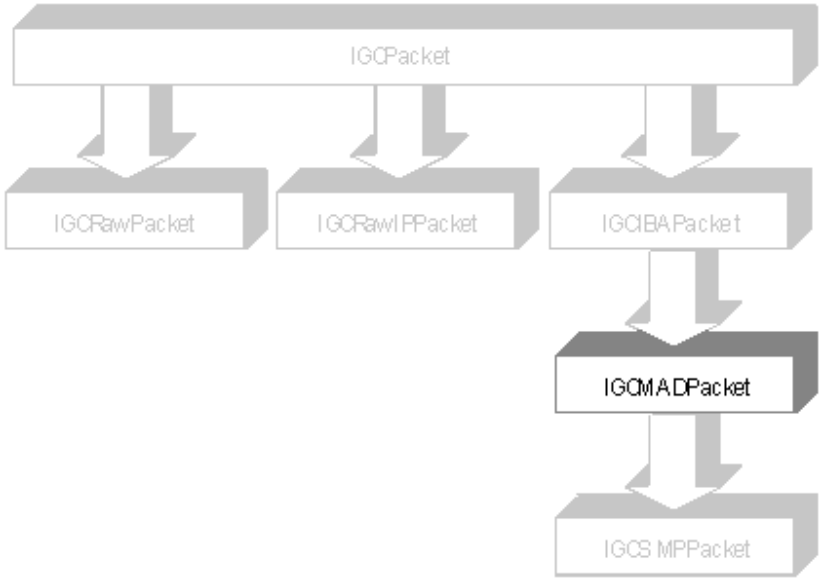
See also *IGCIBAPacket*, Default Constructor and *IGCIBAPacket* Constructor for the Class above

Methods of the IGCMADPacket Class

This class is intended for creating MADs (Management Datagrams). While a MAD packet can also be created using the `IGCIBAPacket` class, it is simpler to use the MAD packet class. The `IGCMADPacket` class is derived from the `IGCIBAPacket` class.

Note that the use of `DeletePacket()` is the recommended method of calling a destructor even though this class has its own destructor. `DeletePacket()` is defined in the base class `IGCPacket` (see *"DeletePacket" on page 2-63*).

Figure 11 IGCMADPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCMADPacket` class:

```
                IGCMADPacket(ig_bool IsGlobal);
virtual         ~IGCMADPacket( void );
```

Inherited Members

The following tables list all inherited members of the `IGCPacket` class that are recommended for direct use (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket *         Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_int32    GetICRC (void) const;
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int32            GetType ( void ) const;
ig_int16            GetVCRC (void) const
virtual ig_bool     HasPayload (void) const
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
void                Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal              Get ( IN ig_int32 prop );
virtual void        Default ( void );
virtual void        CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <igmad.h>
```

IGCMADPacket, Constructor

Call `IGCMADPacket ( ig_bool IsGlobal );`

Description Constructor.

Return Value None

Parameters `IsGlobal`
Boolean value that determines whether the packet carries a global routing header.

See also None

~IGCMADPacket, Destructor

Call `virtual ~IGCMADPacket ( void );`

Description Destructor.

Return Value None

Parameters None

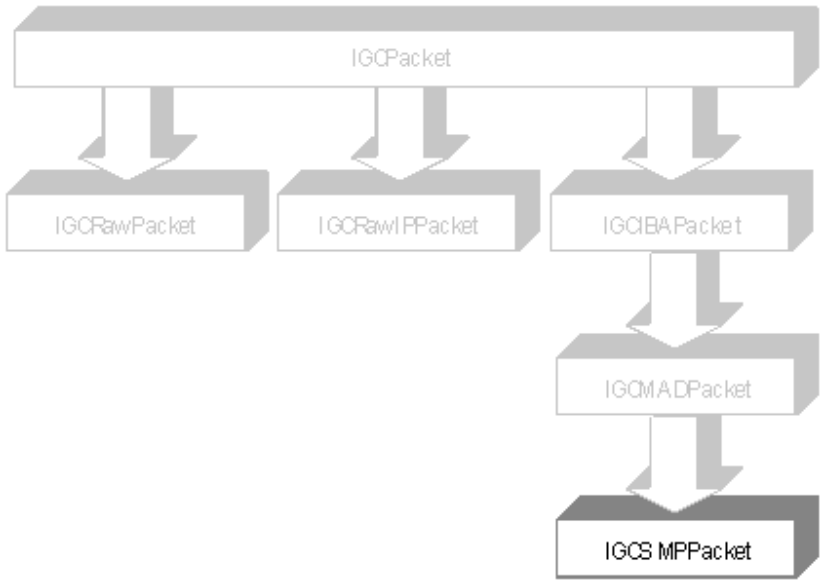
See also None

Methods of the IGCSMPPacket Class

This class is intended for creating SMPs (Subnet Management Packets). While a SMP packet can also be created using the `IGCMADPacket` class, it is simpler to use the SMP packet class. The `IGCSMPPacket` class is derived from the `IGCMADPacket` class.

Note that the use of `DeletePacket()` is the recommended method of calling a destructor even though this class has its own destructor. `DeletePacket()` is defined in the base class `IGCPacket` (see *"DeletePacket" on page 2-63*).

Figure 12 IGCSMPPacket Class



Characteristic Members

The following table lists the characteristic members of the `IGCSMPPacket` class:

```
                IGCSMPPacket();
virtual         ~IGCSMPPacket( void );
```

Inherited Members The following tables list all inherited members of the `IGCPacket` class that are recommended for direct use (see also *"Methods of the IGCObject Class" on page 2-100*):

```
void                AppendBuffer ( OUT IGCBuffer & buffer ) const;
void                AppendPayloadBuffer ( OUT IGCBuffer & buffer ) const;
IGCPacket *         Clone ( void ) const;
void                DeletePacket ( void );
virtual ig_int16    GetActualLength ( void ) const;
virtual ig_int32    GetICRC (void) const;
virtual void        GetPayload (OUT IGCBuffer & bufPld) const;
ig_int32            GetType ( void ) const;
ig_int16            GetVCRC (void) const
virtual ig_bool     HasPayload (void) const
IGCPacket *         NewPacket ( IN & IGCBuffer databuffer );
void                SetPacketLength ( IN ig_int16 length );
void                SetPayload ( IN const & IGCBuffer dataarray );
virtual void        SetPRBSPayloadSize (IN ig_size size);
void                Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal              Get ( IN ig_int32 prop );
virtual void        Default ( void );
virtual void        CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files `#include <igmad.h>`

IGCSMPPacket, Constructor

Call `IGCSMPPacket ( void );`

Description Constructor.

Return Value None

Parameters None

See also None

~IGCSMPPacket, Destructor

Call `virtual ~IGCSMPPacket ( void );`

Description Destructor.

Return Value None

Parameters None

See also None

Methods of the IGCBuffer Class

The class `IGCBuffer` provides a buffer, which contains any number of bytes. The class can be used to represent packets as byte streams or to handle any byte arrays necessary. There are functions to convert packets to buffers and vice versa. The buffer class has also functions to save and load its content.

Characteristic Members

The following table lists all characteristic members of the `IGCBuffer` class:

```
            IGCBuffer ( void );
            ~IGCBuffer ( void );
void        ReadFile ( const char * filename );
void        WriteFile ( const char * filename );
void        SaveFile (const char * filename);
int         Cmp ( IGCBuffer & cmpBuffer ) const;
void        PeekData (ig_size size, ig_int8ptr pRetData) const;
void        Push ( ig_size nBits, const IGCUIntX & inVal );
void        Pop ( ig_size nBits, IGCUIntX & retVal );
void        PopData (ig_size size, ig_int8ptr pRetData);
void        SetAt ( ig_size pos, ig_size nBits, const IGCUIntX & inVal );
ig_size     Size (void) const;
void        GetAt ( ig_size pos, ig_size nBits, IGCUIntX & retVal ) const;
void        Fill ( ig_size size, ig_int8 fillChar );
void        FillRandom (ig_size size);
void        Init ( ig_size size = 0 );
```

Include Files

```
#include <igbuffer.h>
```

IGCBuffer, Constructor

Call `IGCBuffer ( void );`

Description Constructor for the class.

Return Value None

Parameters None

See also None

IGCBuffer, Destructor

Call `~IGCBuffer ( void );`

Description Destructor of the class.

Return Value None

Parameters None

See also None

ReadFile

Call `void ReadFile ( const char * filename );`

Description Reads a file into a buffer.

Return Value None

Input Parameters `filename`
The file to be read.

See also None

WriteFile

Call `void WriteFile ( const char * filename );`

Description Writes the content of a buffer to a file. This deletes the contents of the buffer.

Return Value None

Input Parameters `filename`
The file to be written.

See also None

SaveFile

Call `void SaveFile ( const char * filename );`

Description Saves the content of a buffer to a file without emptying the buffer.

Return Value None

Input Parameters `filename`
The file to be saved.

See also None

Cmp

Call `int Cmp ( IGCBuffer & cmpBuffer ) const;`

Description Compares two buffers bytewise.

Return Value Returns 0 if the buffers are equal, otherwise returns -1 or 1.

Input Parameters `cmpBuffer`
The buffer to compare.

PeekData

Call `void PeekData (ig_size size, ig_int8ptr pRetData) const;`

Description Peeks data from the buffer without modifying its contents.

Return Value None

Input Parameters

`size`
The number of bytes.

`pRetData`
The result.

See also None

Push

Call `void Push ( ig_size nBits, const IGCUIntX & inVal );`

Description Appends bits at the end of the buffer.

Return Value None

Input Parameters

`nBits`
The number of bits to be appended.

`inVal`
The value to which the appended bits should be set.

See also None

Pop

Call `void Pop ( ig_size nBits, IGCUIntX & retVal );`

Description Removes bits starting at the beginning of the buffer.

Return Value None

Input Parameters `nBits`
The number of bits to be removed.

Output Parameters `retVal`
The bits are stored in `retVal`.

See also None

PopData

Call `void PopData (ig_size size, ig_int8ptr pRetData)`

Description Removes the bytestream starting at the beginning of the buffer.

Return Value None

Input Parameters `size`
The number of bytes.

Output Parameters `pRetData`
The bytestream is stored in `pRetData`.

See also None

SetAt

Call `void SetAt ( ig_size pos, ig_size nBits, const IGCUIntX & inVal );`

Description Sets a certain number of bits to a specific value at the position `pos`.

Return Value None

Input Parameters

`pos`
The position in the buffer.

`nBits`
The number of bits to be modified.

`inVal`
The value to which the bits should be set.

See also None

Size

Call `ig_size Size (void) const;`

Description Returns current size of buffer in bytes.

Return Value Size in bytes

Input Parameters None

Output Parameters None.

See also None

GetAt

Call `void GetAt ( ig_size pos, ig_size nBits, IGCUIntX & retVal ) const;`

Description Gets a certain number of bits starting at position `pos`.

Return Value None

Input Parameters

`pos`
The position in the buffer.

`nBits`
The number of bits to be read out.

Output Parameters

`retVal`
The returned value.

See also None

Fill

Call `void Fill ( ig_size size, ig_int8 fillChar );`

Description Fills the entire buffer with the character `fillChar`.

Return Value None

Input Parameters

`size`
Size of the buffer. It is determined by the number of fill characters.

`fillChar`
The character used to fill the buffer.

See also None

FillRandom

Call `void FillRandom ( ig_size size);`

Description Fills the entire buffer with random values.

Return Value None

Input Parameters `size`
Size of the buffer. It is determined by the number of fill characters.

See also None

Init

Call `void Init ( ig_size size = 0 );`

Description Initializes the buffer with a certain size. Note that the buffer is still empty after the call to `Init()`. (`Init()` clears the contents of the buffer).

Return Value None

Input Parameters `size`
The size to which the buffer is initialized. Normally, `Init()` is called to flush the buffer (ensure it is empty). The initial buffer space is 0, this can later be expanded as needed. When called with a size other than 0, the memory is reserved for the specified buffer size. This can make buffer handling more efficient.

See also None

Methods of the IGCVal Class

The various header definitions use various data types for variables (from boolean up to 128 bits for the global route header). For this reason the `IGCVal` class has been created, which can hold all the different types of values. This avoids the need for a separate call for each data type when setting a property with `Set()`. Instead, just one call is necessary, where the `IGCVal` passed with the call holds the correct data type.

Include Files `#include <igval.h>`

IGCVal, Constructor

Call `IGCVal ( void );`

Description Constructor.

Return Value None

Parameters None

See also None

IGCVal, Destructor

Call `virtual ~IGCVal ();`

Description Destructor.

Return Value None

Parameters None

See also None

Constructor by Type

Call `IGCVal ( IGEValType type );`

Description Constructor by type.

Return Value None

Input Parameters `type`
Valid values are:
`igt_INT`
`igt_BOOL`
`igt_UINT8`
`igt_UINT16`
`igt_UINT32`
`igt_STRING`
`igt_UINTX`

See also None

Copy Constructor

Call `IGCVal (const IGCVal & el);`

Description Copy constructor.

Return Value A newly created `IGCVal` object.

Input Parameters `el`
Reference to the object that should be copied.

See also None

Type Conversions

Calls
```
IGCVal (const int);
IGCVal (const ig_bool);
IGCVal (const ig_int8);
IGCVal (const ig_int16);
IGCVal (const ig_int32);
IGCVal (const IGCUIntX &);
IGCVal (const IGCString &);
IGCVal (ig_charcptr);
```

Description Conversions from the data types used.

Return Value The new `IGCVal` class with the appropriate content.

Parameters Parameters are the input values of type int, ig_bool, ig_int8, and so on.

See also None

Const Conversions

Calls

```
operator int ( void ) const;

operator ig_int8 ( void ) const;

operator ig_int16 ( void ) const;

operator ig_int32 ( void ) const;

operator IGCUIntX ( void ) const;

operator ig_bool ( void ) const;

operator IGCString ( void ) const;

operator ig_charcptr ( void ) const;
```

Description Const conversions

Return Value The new `IGCVal` class with the appropriate content.

Parameters None

See also None

Non Const Conversions

Calls

```
operator int ( void );

operator ig_int8 ( void );

operator ig_int16 ( void );

operator ig_int32 ( void );

operator IGCUIntX & ( void );

operator ig_bool ( void );

operator IGCString & ( void );

operator ig_charcptr ( void );

operator ig_int8ptr ( void );

operator ig_charcptr ( void ) const;
```

Description Non const conversions

Return Value The new `IGCVal` class with the appropriate content.

Parameters None

See also None

Assignments

Calls

```
IGCVal & operator = (const int val);

IGCVal & operator = (const ig_int8 val);

IGCVal & operator = (const ig_int16 val);

IGCVal & operator = (const ig_int32 val);

IGCVal & operator = (const IGCUIntX & val);

IGCVal & operator = (const ig_bool val);

IGCVal & operator = (const IGCString & val);

IGCVal & operator = (ig_charcptr val);

IGCVal & operator = (const IGCVal & el);
```

Description

Assignments

Return Value

`IGCVal`
Reference to `IGCVal` object.

Input Parameters

`val`
The assigned value.

`el`
The `IGCVal` to assign.

See also

None

Comparisons

Calls

```
ig_bool operator == (const int val) const;

ig_bool operator == (const ig_int8 val) const;

ig_bool operator == (const ig_int16 val) const;

ig_bool operator == (const ig_int32 val) const;

ig_bool operator == (const IGCUIntX & val) const;

ig_bool operator == (const ig_bool val) const;

ig_bool operator == (const IGCString & val) const;

ig_bool operator == (ig_charcptr pString) const;

ig_bool operator == (const IGCVal & val) const;
```

Description Comparisons

Return Value `ig_bool`
True or false for equal or not equal.

Input Parameters `val`
The respective parameter values are:
integer,
8-bit integer,
16-bit integer,
32-bit integer and so on.

`pString`
`A` pointer to a char buffer (`ig_charcptr` is type defined as const char *). Comparison returns true if either the string in `IGCVal` is equal to the one in `pString`, or the string representation of the value is equal to `pString`. Use with caution!

See also None

Methods of the IGCObject Class

This class is purely virtual and cannot be created by the user. It implements the `Set()` and `Get()` functions for all classes with properties.

The constructor and destructor of this class are protected and not for public use.

Characteristic Members

The following table lists the characteristic members of the `IGCObject` class:

```
void            Set ( IN ig_int32 prop, IN const IGCVal & val );
IGCVal          Get ( IN ig_int32 prop );
virtual void    Default ( void );
virtual void    CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );
```

Include Files

```
#include <igobject.h>
```

Set

Call `void Set ( IN ig_int32 prop, IN const IGCVal & val );`

Description Sets a property to a certain value.

Return Value None

Input Parameters `prop`
The property to be set. The list of properties is dependent on the derived class. For the appropriate list refer to *"Properties and Programmatic Settings" on page 4-1*.

`val`
The value assigned to the property.

See also None

Get

Call `IGCVal Get ( IN ig_int32 prop );`

Description Retreives the value of a specific property.

Return Value `IGCVal` object.

Input Parameters `prop`
The property to be retrieved. The list of properties is dependent on the derived class. For the appropriate list refer to *"Properties and Programmatic Settings" on page 4-1*.

See also None

Default

Call `virtual void Default ( void );`

Description Sets all properties to default values. Refer to *"Properties and Programmatic Settings" on page 4-1*.

Return Value None

Input Parameters None

See also None

CopyProps

Call `virtual void CopyProps ( IN const IGCObject & other, IN ig_bool rwOnly );`

Description Copies `NodeInfo`, `GUIDInfo`, `NodeDescription` and `PortInfo` properties into the generator-owned instances of this class.

Return Value None

Input Parameters

`other`
The object to be copied from.

`rwOnly`
Specifies whether to copy properties that have a set `pr_RO` (read-only permission).

See also None

Methods of the IGCStatus Class

The IGCStatus class contains several information properties that reflect the current status of the connected generator. You pass a reference of an IGCStatus class to the generator and the generator fills the class with the appropriate data.

The properties of this class are listed in *"Properties and Programmatic Settings" on page 4-1*. The Get() function for reading out status property values is derived from IGCObject.

The following table lists the characteristic members of the IGCStatus class:

```
            IGCStatus ( void );
            ~IGCStatus ( void );
ostream     & Print (ostream & o) const;
```

Files `#include <iggenerator.h>`

IGCStatus, Constructor

Call `IGCStatus ( void );`

Description Constructor for the class.

Return Value None

Parameters None

See also None

~IGCStatus, Destructor

Call `~IGCStatus ( void );`

Description Destructor of the class.

Return Value None

Parameters None

See also None

Print

Call `ostream & Print (ostream & o) const;`

Description Prints the status as a text representation to the specified ostream.

Return Value Returns a reference to an `ostream` object with the status information for the connected generator.

Input Parameters `o`
The stream to print into. This provides you with the possibility to print to a file or to stdout.

See also None

Methods of the IGCPacketHandler Class

The class `IGCPacketHandler` provides the methods for the packet handler to manage packets within the handler.

You cannot use the class directly, it is purely virtual and has to be derived. You need to implement versions of the calls `CheckPacket()` and `HandlePacket()`. The generator uses these methods to check if a registered packet handler wants to handle a packet (`CheckPacket()`) and if so, passes the packet to it for handling (`HandlePacket()`). You are free to do whatever is necessary in these two functions.

The following table lists the characteristic members of the `IGCPacketHandler` class:

```
virtual          ~IGCPacketHandler ();
virtual          IGEPacketStatus CheckPacket ( IGCPacket & packet ) = 0;
virtual          IGEPacketStatus HandlePacket ( IGCPacket & packet ) = 0;
IGCGenerator     * GetGenerator ( void ) { return m_pGenerator; }
```

Files `#include <igpackethandler.h>`

~IGCPacketHandler, Destructor

Call `virtual ~IGCPacketHandler ();`

Description Destructor of the class.

Return Value None

Parameters None

See also None

CheckPacket

Call `virtual IGEPacketStatus CheckPacket ( IGCPacket & packet ) = 0;`

Description Called from the generator to check if the packet handler wants to deal with the packet.

Return Value Valid values for the packet status are:

- `REJECT`
  The packet has been rejected by the packet handler.
- `ACCEPT`
  The packet has been accepted by the packet handler.
- `CHANGE`
  The handler modified the packet but still requires another packet handler to handle the packet.

Input Parameters `packet`
The packet to be checked by the packet handler.

See also *HandlePacket* below

HandlePacket

Call `virtual IGEPacketStatus HandlePacket ( IGCPacket & packet ) = 0;`

Description Called from the generator if the call to `CheckPacket()` (see above) returned `ACCEPT`. No other packet handler is called thereafter for this packet.

Return Value The return value is the same as for `CheckPacket`. It is ignored in this release.

Input Parameters `packet`
The packet to be handled by the packet handler.

See also *CheckPacket* above

GetGenerator

Call `IGCGenerator * GetGenerator (void) { return m_pGenerator; }`

Description Returns a pointer to the generator. This can be used to access `PortInfo` or `NodeInfo` structs if needed.

Return Value Pointer to the `IGCGenerator` object.

Parameters None

See also *"IGCNodeInfo Properties" on page 4-6*, *"IGCPortInfo Properties" on page 4-8*

Methods of the IGCPacketHandlerTcl Class

The class `IGCPacketHandlerTcl` lets you register two TCL scripts that handle and check packets. This is a convenience function to help you work with TCL scripts.

NOTE The SMA handler that manages MAD packets is also part of the software distribution. It is provided as a sample in the form of a TCL script.

The following table lists the characteristic members of the `IGCPacketHandlerTcl` class:

```
        IGCPacketHandlerTcl ( char *checkScript, char *handleScript );
virtual ~IGCPacketHandlerTcl ();
```

Files `#include <igpackethandlertcl.h>`

IGCPacketHandlerTcl

Call `IGCPacketHandlerTcl ( char *checkScript, char *handleScript );`

Description Constructor for the class.

Return Value None

Input Parameters `checkScript`
The check script is passed to the class as character pointer.

`handleScript`
The handle is passed to the class as character pointer.

See also None

~IGCPacketHandlerTcl, Destructor

Call `virtual ~IGCPacketHandlerTcl ();`

Description Destructor of the class.

Return Value None

Parameters None

See also None

Methods of the IGCCallBack Class

The class `IGCCallBack` provides the methods to handle callbacks from the API. You cannot use the class directly, it is purely virtual and has to be derived. You need to implement method `Notify()` in your derived class. The generator uses this method to pass the callback data for handling. You are free to do whatever is necessary in this method.

The characteristic members of the `IGCCallBack` class are as follows::

- `IGCCallBack (void)`
- `~IGCCallBack (void)`
- `ig_int32 QueryNotifyMask (void) const`
- `void SetNotifyMask (ig_int32 mask)`
- `virtual IGECBReturn Notify (ig_int32 changeMask, IGCObject & obj, IGCGenerator & generator) = 0`

Files `#include <igcallback.h>`

IGCCallBack, Constructor

Call `IGCCallBack (void);`

Description Default constructor. As `IGCCallBack` is pure virtual, you cannot use this directly. Instead, derive from `IGCCallBack` and implement `Notify`.

Return Value None

Parameters None

See also None

~IGCCallBack, Destructor

Call `virtual ~IGCCallBack ();`

Description Destructor of the class.

Return Value None

Parameters None

See also None

Notify

Call

```
virtual IGECBReturn Notify (ig_int32 changeMask,
                            IGCObject & obj,
                            IGCGenerator & generator) = 0;
```

Description Called from the generator with the callback data.

NOTE Notify is declared protected and cannot be called directly from user programs!

Return Value Valid values are:

- `REJECT (0)` - Pass. nothing changed, pass on data to other callback handlers in chain
- `ACCEPT (1)` - Accept, noone else in the chain gets called
- `CANCEL (2)` - Cancel operation completely (for progress/packet callbacks...)

Input Parameters

`changeMask`
Mask for the changed properties in obj (not for packet callback).

`obj`
The data passed to the callback handler:
`IGCStatus` object for status callbacks,
`IGCProgress` object for progress callbacks
`IGCPacket` object for packet callbacks

`generator`
Reference to the calling generator

See also None

SetNotifyMask

Call `void SetNotifyMask (ig_int32 mask)`

Description Sets the mask for this callback. Each bit represents a property in the data object.

Example:. Set mask to `(1 << IGCStatus::LinkState)` to see link state changes.

Return Value None

Input Parameters `mask`

See also *QueryNotifyMask*

QueryNotifyMask

Call `ig_int32 QueryNotifyMask (void) const`

Description Queries the current notification mask

Return Value The mask.

Parameters None

See also *SetNotifyMask*

Methods of the IGCCallBackTcl Class

The class `IGCCallBackTcl` lets you register a TCL script that handles callback events. This is a convenient function to help you work with TCL scripts.

NOTE The following table lists the characteristic members of the `IGCCallBackTcl` class:

(Constructor) `IGCCallBackTcl (ig_charcptr script);`

Files `#include <igCallBacktcl.h>`

IGCCallBackTcl

Call `IGCCallBackTcl (ig_charcptr script );`

Description Constructor for the class.

Return Value None

Input Parameters `script`
The script is passed to the class as character pointer. It must be the name of a TCL procedure. This must be of the form:

```
proc myScript {pCallBack changeMask pObject pGenerator} { <body> }
```

`pCallBack` is a pointer to the callback object, so you can call `Set/QueryNotifyMask` from within your script.

See also None

Methods of the Error Class

As with all other classes within the generator, the error class `IGCError` can be printed using the C++ stream operator or the method `Print()` (see *"Methods of the IGCGeneratorInfo Class" on page 2-57*). This results in a textual error description readable by humans.

Characteristic Members

The following table lists all characteristic members of the `Error` class:

```
void         Clear ( void );
EErrtype     Error ( void ) const;
             IGCError ( void );
             IGCError ( IN const IGCError & err );
             ~IGCError ( void );
IGCString    GetErrorText ( void ) const;
ostream      & operator << ( ostream & o, const IGCError & theErr );
ostream      & Print (ostream & o) const;
```

Include Files

```
#include <igerror.h>
```

Clear

Call `void Clear( void );`

Description Clears all errors.

Return Value None

Parameters None

See also None

Error

Call `EErrtype Error ( void ) const;`

Description Returns the error code.

Return Value `Eerrtype` object.
Holds the error code. See also *"EErrtype" on page 3-1*.

Parameters None

See also None

IGCError, Constructor

Call `IGCError ( void );`

Description Constructor.

Return Value None

Parameters None

See also None

IGCError, Copy Constructor

Call `IGCError ( IN const IGCError & err );`

Description Copy constructor.

Return Value None

Input Parameters `err`
Reference to an IGCError object.

See also None

IGCError, Destructor

Call `~IGCError ( void );`

Description Destructor.

Return Value None

Parameters None

See also None

GetErrorText

Call `IGCString GetErrorText ( void ) const;`

Description Copy constructor.

Return Value `IGCString` object.
Retrieves the Error string.

Parameters None

See also None

Operator

Call `ostream & operator << ( ostream & o, const IGCError & theErr );`

Description Returns the error to the specified stream in the form of a textual description.

Return Value Reference to the `ostream` object that holds the error text.

Parameters None

See also None

Print

Call `ostream & Print (ostream & o) const;`

Description Prints the content of the class as text representation to the specified `ostream`.

Return Value Returns a reference to an ostream object with the error text.

Input Parameters o
The stream to print into. This provides you with the possibility to print to a file or to stdout.

See also None

Enumeration Definitions

EErrtype

Description Enum over all different error codes. The following list of errors gives detailed descriptions:

Error	Description
IGE_OK = 0	Everything is OK.
IGE_FATAL	Fatal error occurred.
IGE_RANGE	Range checking failed.
IGE_ASSERT	Assertion failed. Usually an unrecoverable error.
IGE_OUTOFMEM	The application has run out of memory.
IGE_INVALIDHANDLE	The handle you are using is invalid.
IGE_SYNTAX	Syntax error while parsing input parameters (for example pattern terms).
IGE_NOTINITIALIZED	The object needs initializing before use.
IGE_FILENOTFOUND	The specified file could not be found.
IGE_TESTFAILED	The test failed.
IGE_WARNING	Is used to transport a warning message.
IGE_INVALIDPACKET	Packet parsing failed.
IGE_FWERROR	Generator firmware encountered an error.
IGE_WIN32	Error generated by WIN32 calls.
IGE_UNDEFPROP	Undefined property.
IGE_INVALIDTYPE	Invalid type used.
IGE_BUFFERUNDERRUN	Buffer too small. Emptied while reading out data.
IGE_VERSION	Version mismatch. Upgrade to the newest hardware version.
IGE_GENERAL	General failure (none of the above).

IGCGenerator::IGEPropName

Description Enumerated integer for all property values that can be set within the generator. For a description of the properties, see *"Properties and Programmatic Settings" on page 4-1*.

IGCPacket::IGEPropName

Description Enumerated integer for all property values that can be set within the base class `IGCPacket` (see also *"Properties and Programmatic Settings" on page 4-1*).

IGCVal::Opcode

Description Enumerated integer offering a choice of available opcodes. For the exact definition, see the opcode overview below:

Reliable Connection	Unreliable Connection	Reliable Datagram	Unreliable Datagram
RC_SENDFirst=0x0	UC_SENDFirst=0x20	RD_SENDFirst=0x40	0x60-0x63 Reserved for UD
RC_SENDMiddle	UC_SENDMiddle	RD_SENDMiddle	UD_SENDOnly=0x64
RC_SENDLast	UC_SENDLast	RD_SENDLast	UD_SENDOnlyImm
RC_SENDLastImm	UC_SENDLastImm	RD_SENDLastImm	0x66 0x7f Reserved for UD
RC_SENDOnly	UC_SENDOnly	RD_SENDOnly	0x80-0xbf Reserved
RC_SENDOnlyImm	UC_SENDOnlyImm	RD_SENDOnlyImm	0xc0-0xff Manufacturer specific opcodes
RC_RDMAWRITEFirst	UC_RDMAWRITEFirst	RD_RDMAWRITEFirst	
RC_RDMAWRITEMiddle	UC_RDMAWRITEMiddle	RD_RDMAWRITEMiddle	
RC_RDMAWRITELast	UC_RDMAWRITELast	RD_RDMAWRITELast	
RC_RDMAWRITELastImm	UC_RDMAWRITELastImm	RD_RDMAWRITELastImm	
RC_RDMAWRITEOnly	UC_RDMAWRITEOnly	RD_RDMAWRITEOnly	
RC_RDMAWRITEOnlyImm	UC_RDMAWRITEOnlyImm	RD_RDMAWRITEOnlyImm	
RC_RDMAREADRequest	0x2c-0x3f Reserved for UC	RD_RDMAREADRequest	

Reliable Connection	Unreliable Connection	Reliable Datagram	Unreliable Datagram
RC_RDMAREADresponseFirst		RD_RDMAREADresponseFirst	
RC_RDMAREADresponseMiddle		RD_RDMAREADresponseMiddle	
RC_RDMAREADresponseLast		RD_RDMAREADresponseLast	
RC_RDMAREADresponseOnly		RD_RDMAREADresponseOnly	
RC_Acknowledge		RD_Acknowledge	
RC_AtomicAcknowledge		RD_AtomicAcknowledge	
RC_CmpSwap		RD_CmpSwap	
RC_FetchAdd		RD_FetchAdd	
0x15-0x1f Reserved for RC		0x55-0x5f Reserved for RD	

Properties and Programmatic Settings

The following lists of properties are used to program the E2953A. You can set all properties directly. Some settings have a generic impact on the behavior of the generator, these are listed under *Generator Properties* below. Certain settings can be made with any packet, these are listed under *"Status Properties" on page 4-3*.

Some values necessary for the generation of packet headers are supplied by the global property lists or are set by a subnet manager (source is the local id). These values can be preset in a packet using the call `PacketInit()` in the class generator.

Generator Properties

Generator properties control the behavior of the generator. They also influence the content of headers of outgoing packets. The prefix `IG__` stands for generic generator properties. Header properties have an appropriate prefix (`LRH_` or `AECK`). The `IGCGenerator` method `PacketInit()` initializes packets with the correct header information.

Table 8 Generator Property List (IGCGenerator::Prop)

Property Name	Range	Default	Description
PRBSSeed	0 – 2^{11}	1	Starts a seed of the internal PRBS for data payload generation. A value of 0 results in all 0s for the payload.
BADPacketDiscard	0 – 1	1 = discard	Discards or keeps invalid packets on receive. Works for packets with bad ICRC (bad VCRC and EBP are discarded in HW).
TransmitRepeatCounter	0 – 65535	1 (0 means infinity)	Sets a counter on how often the transmit memory is to be repeated.
RepeatCounter1, RepeatCounter2, RepeatCounter3	0 – 2^{16}	1 (0 means infinity)	Holds a value for the repeat line counter of the block memory. The packet that selects one of the counters get repeated countervalue number of times. RepeatCounter0 is not accessible by the user.
PSNStartValue	24 bit	1	Packet sequence number start value. If the automatic packet sequence number generation is enabled, this value is taken as the start value (see IGP_AutoCalculatePSN in *"Generic Packet Properties" on page 4-12*).
CodeGroup	0 – 2^{10}	0	This code group is used if the error insertion method for the packet selects an invalid code group as an error to be inserted into the packet.

Status Properties

The following list of properties determine the current state of the connected generator. They stem from the class `IGCStatus`.

Table 9 IGCStatus Property List (IGCStatus::EPropName)

Property Name	Range	Default	Description (if = 1)
`TransmitRunning`	0 - 1	0	The transmit memory is currently sending packets.
`TransmitFinished`	0 - 1	0	The transmit memory has completed sending all packets.
`TransmitError`	0 – 1	0	The link was down when sending packets or it was downed while sending packets from the transmit memory. The transmit memory is automatically switched to stop state.
`TransmitWaitTriggerIn`	0 – 1	0	The next packet in the transmit memory is waiting for the trigger-in signal to commence.
`TransmitWaitDelay`	0 – 1	0	The transmit memory is currently waiting for the delay counter to finish.
`TransmitWaitCredits`	0 – 1	0	The transmit memory packet stream is currently waiting for new credits on the virtual lane of the packet next in line.
`TransmitWaitStep`	0 – 1	0	The transmit memory packet stream is currently waiting for a software data strobe or a pattern action event.
`TransmitWaitLink`	0 – 1	0	The transmit memory packet stream is waiting for the IB link to establish before it can start. If the link is interrupted while the transmit memory is running, `TransmitError` is signaled.
`SendRunning`	0 – 1	0	The send buffer is currently active and trying to send a packet.
`SendFinished`	0 – 1	0	The send buffer has successfully sent out a packet.

Property Name	Range	Default	Description (if = 1)
SendError	0 – 1	0	The link was down when sending packet or it was downed while sending packet from buffer. The send buffer automatically switches back to stop mode.
SendWaitTriggerIn	0 – 1	0	The send buffer is waiting for an external trigger event.
SendWaitDelay	0 – 1	0	The send buffer is waiting for the delay counter to finish.
SendWaitCredits	0 – 1	0	The send buffer is currently waiting for credits for its packet.
SendWaitStep	0 – 1	0	The send buffer is currently waiting for a pattern action or a software data strobe.
SendWaitLink	0 – 1	0	The send buffer packet is waiting for the IB link to establish before it can start. If the link is interrupted while the send buffer is currently sending the packet, TransmitError is signaled.
LinkTrainingState	--	LINKTRAINSTATE_DISABLED	Current state of the link training state machine. The following results are possible: LINKTRAINSTATE_DISABLED LINKTRAINSTATE_POLLACTIVE LINKTRAINSTATE_POLLQUIET LINKTRAINSTATE_CFGDEBOUNCE LINKTRAINSTATE_CFGRCVRCFG LINKTRAINSTATE_CFGWAITRMT LINKTRAINSTATE_CFGIDLE LINKTRAINSTATE_LINKUP LINKTRAINSTATE_RECRETRAIN LINKTRAINSTATE_RECWAITRMT LINKTRAINSTATE_RECIDLE LINKTRAINSTATE_SLEEPDELAY LINKTRAINSTATE_SLEEPQUIET

Property Name	Range	Default	Description (if = 1)
LinkState	--	LINKSTATE_DOWN	Current state of the link state machine. The following results are possible: LINKSTATE_DOWN LINKSTATE_ARM LINKSTATE_ACTIVE LINKSTATE_INIT LINKSTATE_ACTIVEDEFER
LaneSkew	--	0	Receiver Lane Skew Status: Bit0-3: LaneA Bit4-7: LaneB Bit8-11: LaneC Bit12-15: LaneD The unit of the Skew is Symbol Times.

IGCNodeInfo Properties

The `IGCNodeInfo` properties are determined by the list of values kept in the struct `NodeInfo` for each InfiniBand port. A complete description is available in the InfiniBand Specification, Section 14.2.5.3.

Table 10 List of IGCNodeInfo Properties

Property Name	Range	Default	Access	Description
`BaseVersion`	$0 - 2^8$	1	RO	Supported MAD Base Version
`ClassVersion`	$0 - 2^8$	1	RO	Supported Subnet Management Class Version
`Type`	$0 - 2^8$	1	RO	The default is to emulate a channel adapter (=1);
`NumPorts`	$0 - 2^8$	1	RO	Number of physical ports on this node
`Reserved32`	$0 - 2^{64}$	0	RO	Reserved, shall be zero
`GUID`	$0 - 2^{64}$	0	RO	GUID of the end node
`PortGUID`	$0 - 2^{64}$	0	RO	GUID of this port itself
`PartitionCap`	$0 - 2^{16}$	1	RO	Entries in partition table
`DeviceID`	$0 - 2^{16}$	0x2953	RO	Assigned by manufacturer
`Revision`	$0 - 2^{32}$	1	RO	Device Revision
`LocalPortNum`	$0 - 2^8$	1	RO	Link Port number for this SMP
`VendorID`	$0 - 2^{24}$	0x15bc	RO	Device vendor (IEEE)

IGCNodeDescription Properties

The `IGCNodeDescription` properties are determined by the list of values kept in the struct `NodeDescription` for each InfiniBand port. The complete description is listed in the InfiniBand Specification, Section 14.2.5.2.

Table 11 List of IGCNodeDescription Properties

Property Name	Range	Default	Access	Description
`NodeString`	512 bit	"Agilent E2953A 1x Generator for InfiniBand"	RO	Unicode string to describe the node in text format.

IGCGUIDInfo Properties

The `IGCGUIDInfo` properties are determined by the list of values kept in the struct `GUIDInfo` for each InfiniBand port. The complete description is listed in the InfiniBand Specification, Section 14.2.5.5.

Table 12 List of IGCGUIDInfo Properties

Property Name	Range	Default	Access	Description

`GUID0`
`GUID1`
`GUID2`
`GUID3`
`GUID4`
`GUID5`
`GUID6`
`GUID7`
	0 - 2^{64}	0	RW	Eight GUID blocks to be assigned to this port.

IGCPortInfo Properties

The `IGCPortInfo` properties are determined by the list of values kept in the struct `PortInfo` for each InfiniBand port. The complete description is listed in the InfiniBand Specification, Section 14.2.5.6.

Table 13 List of IGCPortInfo Properties

Property Name	Range in bits	Default	Access	Description
`M_Key`	64	0x0	RW	Management key.
`GidPrefix`	64	0xfe800000: 0x00000000	RW	GID Prefix for this port.
`LID`	16	0xffff	RW	Base LID for this port.
`MasterSMLID`	16	0x0	RW	Base LID of Master SM.
`CapabilityMask`	32	0x00000200	RO	Supported Capabilities of this node. See IB Spec for details.
`DiagCode`	16	0x0	RO	Diagnostic code.
`M_KeyLeasePeriod`	16	0x0	RW	Number of seconds for `M_Key` Lease period.
`LocalPortNum`	8	0x1	RO	The link port number this SMP came in.
`LinkWidthEnabled`	8	0x1	RW	Enabled Link Width, see IB Spec for details.
`LinkWidthSupported`	8	0x1	RO	Supported Link Width, see IB Spec for details.
`LinkWidthActive`	8	0x1	RO	Currently active Link Width, see IB Spec for details.
`LinkSpeedSupported`	4	0x1	RO	Supported Link Speed, see IB Spec for details.
`PortState`	4	0x0	RW	Current Port State, see IB Spec for details.

Property Name	Range in bits	Default	Access	Description
PortPhysicalState	4	0x5	RW	Current Port physical state, see IB Spec for details.
LinkDownDefaultState	4	0x0	RW	Link Down State. Only valid transitions are valid if writing this field.
M_KeyProtectBits	2	0x0	RW	Defines the level of protection.
Reserved274	3	0	RO	Reserved, shall be zero.
LMC	3	0x0	RW	LID mask for multipath support.
LinkSpeedActive	4	0x1	RO	Current active link speed.
LinkSpeedEnabled	4	0x1	RW	Enabled Link Speed.
NeighborMTU	4	0x1	RW	Active Maximum MTU.
MasterSMSL	4	0x0	RW	The administrative SL of the Master.
VLCap	4	0x2	RO	Supported Virtual Lanes.
Reserved300	4	0	RO	Reserved, shall be zero.
VLHighLimit	8	0x0	RW	Limit of high priority component.
VLArbitrationHighCap	8	0x0	RO	VL pairs for high priority.
VLArbitrationLowCap	8	0x0	RO	VL pairs for low priority.
Reserved328	4	0	RO	Reserved, shall be zero.
MTUCap	4	0x5	RO	Maximum MTU supported.
VLStallCount	3	0x0	RW	Number of sequential packets dropped to enter the VLStalled state.
HOQLife	5	0x1f	RW	Time a packet can live at head of VL queue.

Property Name	Range in bits	Default	Access	Description
OperationalVLs	4	0x2	RW	VL operational at this port.
PartitionEnforcemetInbound	1	0x0	RW	Support for optional partition enforcement (receiving packets).
PartitionEnformcementOutbound	1	0x0	RW	Support for optional partition enforcement (transmitting packets).
FilterRawPacketInbound	1	0x0	RW	Support for optional raw packet enforcement (receiving packets).
FilterRawPacketOutbound	1	0x0	RW	Support for optional raw packet enforcement (transmitting packets).
M_KeyViolations	16	0x0	RW	Number of SMP packets with invalid M_Keys.
P_KeyViolations	16	0x0	RW	Number of SMP packets with invalid P_Keys.
Q_KeyViolations	16	0x0	RW	Number of SMP packets with invalid Q_Keys.
GUIDCap	8	0x0	RO	Number of supported GUID entries.
Reserved408	3	0	RO	Reserved, shall be 0.
SubnetTimeOut	5	0x1f	RW	Maximum expected subnet propagation delay.
Reserved416	3	0	RO	Reserved, shall be 0.
RespTimeValue	5	0x1f	RO	Maximum time between SMP reception and associated response.
LocalPhyErrors	4	0x0	RW	Threshold value for marginal link errors.
OverrunErrors	4	0x0	RW	Threshold value for overrun errors.

Packet Properties

The following lists of properties are used to set up a single packet. The packet can then be passed to an object of type `generator` and can either be sent immediately, or it can be programmed into the transmit memory.

The following lists of properties are part of `IGCPacket::IGEPropName` or `IGCIGAPacket::IGEPropName`. These properties are divided into a general list of properties (prefix `IGP_`) and the property lists that belong to the various headers (prefixes `LRH_`, `ATEH_` and so on).

The properties are different enums, but because they can all be mapped to an integer and the implementation ensures the values are distinct, the same '`set/get`' function can be used for setting/getting all properties.

Generic Packet Properties

Packet properties consist of pure packet properties as well as header and payload settings necessary to make up an InfiniBand packet.

Table 14 Property List (Generic Portion)

Property Name	Range	Default	Description
IGP_InterPacketDelayOffset	0 - 3	0	Inter Packet Delay (before this packet). The real value 'd' for the delay is calculated using the following formula: $d = offset + 8^{exponent} - 1$
IGP_InterPacketDelayExponent	0 - 7	0	See property above.
IGP_Repeat	0 - 3	0	Defines the repeat counter to be taken for this packet. A value of 0 means a fix repeat value of 1. All other repeat counter values can be set as generator properties.
IGP_InsertError	0 – 6	0 = no error	Code for the error to be inserted at the end of the packet. For a detailed list of error codes see error list (next table).
IGP_BadICRC	0 – 1	0	Create bad ICRC.
IGP_BadVCRC	0 – 1	0	Create bad VCRC.
IGP_IgnoreCredit	0 – 1	0	Ignore Credit status (send anyway).
IGP_AutoCalculatePSN	0 - 1	1 = autocalculate	Calculate the PSN automatically starting with a generic start value out of a register.
IGP_PayloadSize	0 - 4096	0	Payload Size in Bytes.
IGP_UsePRBS	0 - 1	0 =no PRBS	Use PRBS instead of programmed payload.
IGP_WaitTriggerIn	0 - 1	0 = do not wait	Wait for trigger in.
IGP_AssertTriggerOut	0 – 1	0 = do not assert	Assert trigger out (at beginning of packet before inter packet delay starts).
IGP_WaitStep	0 - 1	0 = do not wait	Wait for a TransmitStep event (puts a packet on hold until the user issues a TransmitStep call or a pattern term asserts this signal, which allows waiting for software controlled acknowledges or specific external events (via pattern term).

Error insertion using an error code is done at the end of a packet. The worst test case for the receiving decoder occurs when any of the symbols listed in the following table are received, with the exception of EGP (end of good packet).

Sending four running disparity errors within 16 symbol clocks amounts to a check whether the InfiniBand link automatically reinitializes (see the table below).

Table 15 Error Code Table for Generic Packet Properties

Appended Symbol	Value	Description
EGP	0	Ends a packet with the 'end of good packet' symbol.
EBP	1	Ends a packet with the 'end of bad packet' symbol.
SLP	2	Ends a packet with the 'start of link packet' symbol.
SDP	3	Ends a packet with the 'start of data packet symbol'.
Invalid Code Group	4	Sends out an invalid code group instead of EGP. The invalid code group can be specified in a generator property.
Running Disparity Error	5	Inserts a running disparity error.
Running Disparity Error Burst	6	Inserts 4 running disparity errors spread out over 16 symbols.
Reserved	7	Not used.

Local Routing Header Properties

These are also part of the `IGCPacket::IGEPropName` property values. The following lists of properties give an overview of what can be set within an InfiniBand packet. The lists follow very much the specification for InfiniBand headers.

Table 16 Part of IGCPacket::IGEPropName: Local Routing Header Props

Property Name	Range	Default	Description
`LRH_VL`	0 - 15	0	Virtual Lane.
`LRH_LVer`	0 - 15	0	Link Version.
`LRH_SL`	0 - 15	0	Service Level.
`LRH_Resv12`	0 - 4	0	Reserved_lrh1.
`LRH_LNH`	0 - 4	Precalculated according to the packet type.	Link Next Header. Can be overwritten by the user.
`LRH_DLID`	0 - 65535	0	Destination Local ID.
`LRH_Resv32`	0 - 31	0	`Reserved_lrh2`
`LRH_PktLen`	0 - 2047	0	Packet Length
`LRH_SLID`	0 – 65535	0	Source Local ID. This property gets set with the correct value from the generator using the method `IGCGenerator::PacketInit`.

Global Routing Header Properties

This list of properties shows what you can set in the global routing header (if present).

Table 17 IGCIBAPacket::IGEPropName: Global Routing Header Props

Property Name	Range	Default	Description
GRH_IPVer	0 - 15	6	IP Version.
GRH_TClass	0 - 255	0	Traffic Class.
GRH_FlowLabel	$0 - 2^{20}$	0	Flow Label.
GRH_PayLen	0 – 65535	0	Payload Length.
GRH_NxtHdr	0 - 255	Pre-calculated	Next Header.
GRH_HopLmt	0 - 255	User	Hop Limit.
GRH_SGID	$0 - 2^{128}$	0	Source GID. This property gets set from the generator using the method `IGCGenerator::PacketInit`.
GRH_DGID	$0 - 2^{128}$	0	Destination GID.

Base Transport Header Properties

This header is present in all packets.

Table 18 IGCIBAPacket::Prop: Base Transport Header Properties

Property Name	Range	Default	Description
BTH_OpCode	IGCVal::Opcode	0	Opcode. To use a reserved opcode specify the value as ig_int32.
BTH_SE	0 - 1	0	Solicited Event.
BTH_M	0 - 1	User	Migration State.
BTH_PadCnt	0 - 4	Pre-calculated	Pad Count.
BTH_TVer	0 - 16	0	Transport Header Version.
BTH_P_KEY	0 - 65535	0	Partition Key.
BTH_Reserved32	0 - 255	0	Reserved (variant).
BTH_DestQP	$0 - 2^{24}$	0	Destination QP.
BTH_A	0 - 1	0	Acknowledge Request.
BTH_Reserved65	0 - 128	0	Reserved.
BTH_PSN	$0 - 2^{24}$	0	Packet Sequence Number.

Extended Transport Header Fields

Depending on the opcode, different types of extended header fields are present. These fields and the appropriate property values are listed below. The Reliable Datagram Extended Transport Header (RDETH) is always present if the packet is part of a reliable datagram message.

Table 19 Reliable Datagram Extended Transport Header (RDETH)

Property Name	Range	Default	Description
RDETH_Reserved0	0 - 255	0	Reserved.
RDETH_EECnxt	0 – 2^{24}	0	EE-context

The Datagram Extended Transport Header (DETH) is present in every packet that is part of a datagram request message.

Table 20 Datagram Extended Transport Header (DETH) Properties

Property Name	Range	Default	Description
DETH_Q_Key	0 – 2^{32}	0	Queue Key.
DETH_Reserved32	0 - 255	0	Reserved.
DETH_SrcQP	0 – 2^{24}	0	Source QP.

The RDMA Extended Transport Header is present in the first packet of a RDMA (Remote Direct Memory Access) request message.

Table 21 RDMA Extended Transport Header Property List

Property Name	Range	Default	Description
RETH_VA	0 – 2^{64}	0	Virtual Address.
RETH_R_Key	0 – 2^{32}	0	Remote Key.
RETH_DMALen	0 – 2^{32}	0	DMA Length.

The Atomic Extended Transport Header is present in atomic request messages.

Table 22 Atomic Extended Transport Header (AtomicTEH) Property List

Property Name	Range	Default	Description
AtomicETH_VA	0 – 2^{64}	0	Virtual Address.
AtomicETH R_Key	0 – 2^{32}	0	Remote Key.
AtomicETH _SwapDt	0 – 2^{64}	0	Swap (or Add) Data.
AtomicETH _CmpDt	0 – 2^{64}	0	Compare Data.

The ACK Extended Transport Header is present in all ACK packets, including the first and last packet of a message for RDMA Read Response packets.

Table 23 ACK Extended Transport Header (AETH) Property List

Property Name	Range	Default	Description
AETH_Syndrome	0 - 255	0	Syndrome
AETH_MSN	0 – 2^{24}	0	Message Sequence Number.

The Atomic ACK Extended Transport Header is present in all Atomic ACK packets.

Table 24 Atomic ACK Extended Transport Header Property List

Property Name	Range	Default	Description
AtomicAckETH_OrigRemDt	0 – 2^{64}	0	Original Remote Data.

The Immediate Data Extended Transport Header is present in the last packet of a request with immediate data.

Table 25 Immediate Data Extended Transport Header Property List

Property Name	Range	Default	Description
ImmDt	0 – 2^{32}	0	Immediate Data.

The payload for the packet either comes out of a PRBS (generic packet property) or is handed to the software as a pointer to a data array. Only one behavior is needed to specify the payload size. This is controlled by a generic packet property.

The software calculates the invariant CRC and the variant CRC automatically. Generic packet properties are available to let you create incorrect CRCs.

Index

Index

Publication Number: 5988-5061EN

Agilent Technologies