Gemini Observatory ICD50

GIAPI C++ Language Glue API

Version 11

Arturo Núñez, Kim Gillies

GIAPIC++ICD50-02042008/10
Created: February 4, 2008
Modified: January 28, 2014



Change Record

Version	Date	Author	Notes
1	February 28, 2008	Arturo Núñez, Kim Gillies	• First Version
2	July 24, 2008	Arturo Núñez	• Second Version. Includes changes agreed on visit to HIA in March 2008. Minor updates to API.
3	August 29, 2008	Arturo Núñez	• Third Version. Incorporated comments and corrections from Jennifer Dunn after second release
4	October 21, 2008	Arturo Núñez	• Fourth Version. Added exceptions to the GIAPI
5	February 3, 2009	Arturo Núñez	• Replaced `char*` arguments in the `ServiceUtil` class with standard C++ strings.
6	April 9, 2009	Arturo Núñez	• Added timeout to `getProperty()` method. • Added new `TimeoutException`. • Replaced `char*` with standard C++ string arguments in methods of the `StatusUtil`, `HandlerResponse`, `GeminiUtil`, `EpicsStatusItem`, `DataUtil`, `Configuration`, and `CommandUtil` classes. • Added utility methods to the `EpicsStatusItem` interface to facilitate data decoding. • Added section about the `GiapiUtil` class.
7	December 21, 2009	Arturo Núñez	• Added reference to float support for status items • Added timeout to `getTcsContext()` call.
8	February 18, 2010	Arturo Núñez	• Removed unnecessary `ALL` level for system logging service.
9	February 23, 2011	Nicolas A. Barriga	• Added info on `gmp.properties` config file.
10	February 03, 2012	Nicolas A. Barriga	• Added info on Engineering commands
11	January 17, 2014	Carlos Quiroz	• Added information about the `getChannel` method

Table of Contents

1 Introduction 1

1.1 Document Purpose 1

1.2 Intended Readership 1

1.3 Conventions 1

1.4 Acronyms 2

1.5 Reference Materials 2

2 Overview of GIAPI C++ Language Glue API 3

3 Library Usage 4

3.1 Compiler and build tools 4

3.2 Dependencies 5

3.3 Accessing the library C++ public headers 5

3.4 Linking with the library 5

3.5 General conventions used in the C++ Language Glue API 6

3.6 Exceptions in the C++ Language Glue API 6

4 Providing Status to Gemini 7

4.1 General overview 7

4.2 Status Item initialization 9

4.3 Setting Values to status items 10

4.3.1 Setting values to regular status items 10

4.3.2 Setting Alarms 11

4.3.3 Setting Health 12

4.4 Posting status to Gemini 12

5 Receiving Sequence Commands from Gemini 13

5.1 General Overview 13

5.2 Sequence Command Handlers 15

5.3 Subscribing to Sequence Commands other than Apply 18

5.3.1 Arguments in Sequence Commands 20

5.4 Subscribing to the Apply Sequence Command 21

5.5 Updating Gemini with Action Status 21

6 Interacting with Gemini Systems 23

6.1 General Overview 23

6.2 Receiving Status Information from a Gemini System 23

6.3 Access to current EPICS values 27

6.4 Provide Wavefront Sensor corrections to the PCS 27

6.5 Access to the TCS Context 28

7 Using Gemini Services 29

7.1 General Overview 29

7.2 Logging Service 29

7.2.1 Process Logging 29

7.2.2 System Logging 29

7.3 Access to Observatory Time 30

7.4 GIAPI Configuration Properties 30

8 Providing Science Data to Gemini *31*

8.1 General overview 31

8.2 Sending Observations Events with GIAPI 32

8.3 Ancillary Files and GIAPI 33

8.4 Support for Intermediate Files 33

9 Error Handling and Fault Tolerance in the GIAPI C++ Language Glue *34*

9.1 General Overview 34

9.2 Error Handlers for GMP communication problems 35

9.2.1 Object-oriented approach to deal with GMP communication problems 35

9.2.2 A procedural approach to deal with GMP communication problems 36

10 Exceptions in the GIAPI C++ Language Glue *36*

10.1 Exception Classes in the GIAPI C++ Language Glue 36

1 Introduction

This document provides a description of the C++ Programming Language Application Programmer Interface for the Gemini Instrument Application Programmer Interface (GIAPI). The GIAPI concept was introduced in 2004 to support the second generation of Gemini instruments, the products of the Aspen instrument process.

The Aspen instrument documentation tree is shown in Figure 1. The document you are reading is a companion document to the top-level document *Aspen GIAPI Design and Use* [1]. GIAPI Design and Use is the document that describes how the GIAPI works and how builders should use the library to integrate with Gemini software and hardware systems. Requirements and patterns for its use are described to clarify how responsibilities are balanced between Gemini and the instrument builder.

The GIAPI provides a client integration library in two computer programming languages: C++ and Java. This document is an Interface Control Document (ICD) describing how instrument code written in the C++ programming language uses the GIAPI.

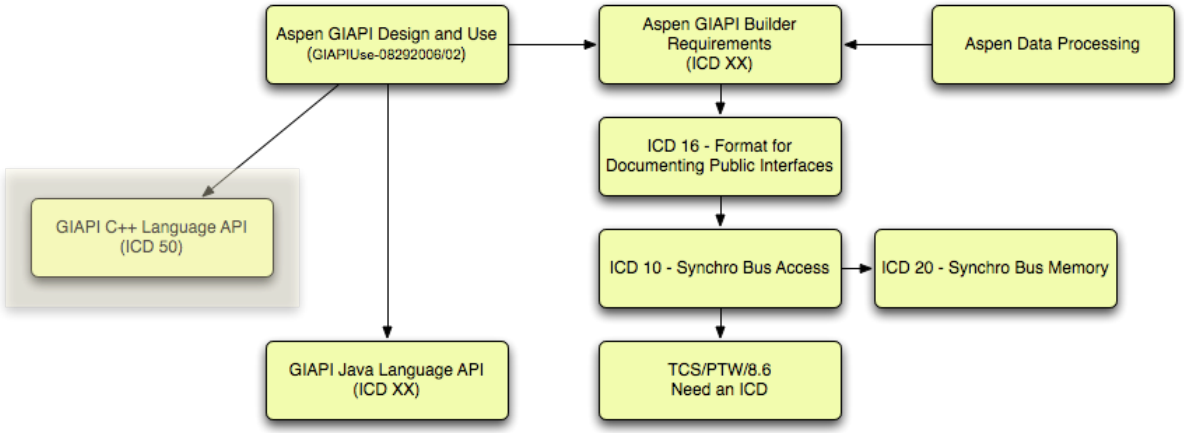


Figure 1: GIAPI Documentation Tree

1.1 Document Purpose

The purpose of this document is to present the C++ GIAPI Language API. This document does the following:

- Provides C++ compiler tool standards.
- Provides descriptions of the GIAPI C++ implementation methods.
- Provides information on how to use the GIAPI C++ client integration library.

This document does not rehash the critical information in references [1] and [2]. It is assumed that the reader has read these other documents and understands their content.

1.2 Intended Readership

The intended audience for this document is *groups who are writing software or design review documents for Aspen instruments*. The document is targeted specifically towards C++ programmers or other specialists who understand software and who must design or implement the TLC for a Gemini Aspen instrument.

1.3 Conventions



The GIAPI is still under development in some areas and things that are expected to undergo some changes are marked like this paragraph with a yellow exclamation point. There are not many of these situations in this document.

Code examples and individual methods are written in a fixed-width font like this: `unsubscribeToStatus`.

1.4 Acronyms

ACM	Action Command Model
CMS	C++ Messaging Service
DHS	Data Handling System
GIAPI	Gemini Instrument Application Programmer Interface
GMP	Gemini Master Process
GSDN	Gemini Data Storage Network
ICD	Interface Control Document
JMS	Java Message Service
PCS	Primary Control System
TCS	Telescope Control System
TLC	Top Level Computer
WCS	World Coordinate System

1.5 Reference Materials

[1] *Guidelines for Designing Gemini Aspen Instrument Software*, Kim Gillies, AspenSoft-03072004-6.

[2] *Aspen GIAPI Design and Use*, Kim Gillies, Arturo Núñez, GIAPIUse-08292006-02.

[3] FITS Standard Specification: http://archive.stsci.edu/fits/fits_standard/fits_standard.html.

[4] FITSIO Home Page, http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html

[5] TCS/PTW/8.6, World Coordinates, Part 1: Astrometry, P.T. Wallace, RAL.

[6] Java Message Service Home Page, http://java.sun.com/products/jms/

[7] Apache log4xx Home Page, http://logging.apache.org/log4cxx/index.html

[8] The boost C++ Libraries Home Page http://www.boost.org/

[9] The CppUnit framework Home Page, http://cppunit.sourceforge.net/cppunit-wiki

[10] Apache ActiveMQ Home Page, http://activemq.apache.org/

[11] Apache ActiveMQ CPP Home Page, http://activemq.apache.org/cms/

2 Overview of GIAPI C++ Language Glue API

The GIAPI C++ Language glue API (also referred in this document as the GIAPI C++ API or simply as the C++ API) is one of the Language binding options used by instrument code to integrate with Gemini. The C++ API is implemented as a shared library in C++. Figure 2 shows the relationship between the C++ API and the instrument code. The API sits between the instrument code and the Gemini Master Process (GMP).

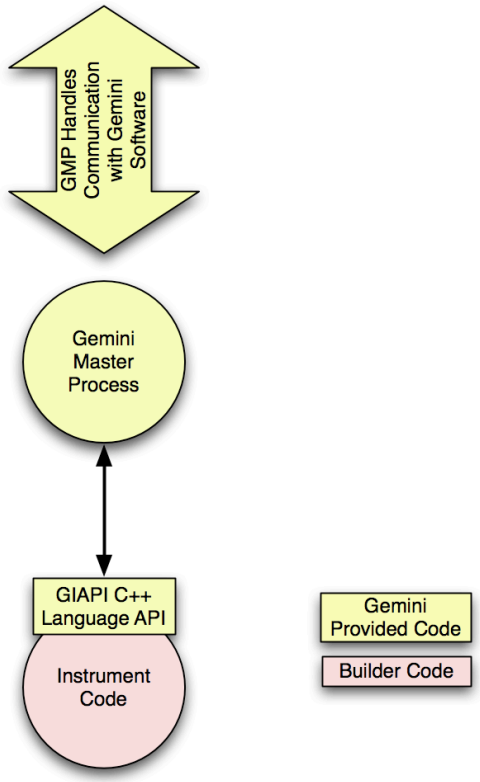


Figure 2: Relationship between C++ Language API and instrument code

The GIAPI C++ API library is included with any process that the builder creates that must communicate with Gemini. Which processes in the instrument must link with and use the C++ API is an instrument design decision.

The communication protocol between the C++ API and the GMP is encapsulated within the GIAPI implementation.

The design of the C++ Language glue is based on a collection of utility classes, as shown in Figure 3. These utility classes are organized by the functionality they provide and the description of the functionality in [2]. Each utility class provides static methods the instrument code can call to access the integration functionality offered by the GIAPI framework.

The utility classes provided by the C++ API are:

- StatusUtil: This class provides the mechanisms needed for instrument code to provide status information (including alarms and health information) to Gemini. Details about this class are presented in section 4 in this document, *Providing Status to Gemini*.
- CommandUtil: Mechanisms to receive and handle sequence commands are provided by the *CommandUtil* class. With this class, instrument code can receive sequence commands and provide completion information to Gemini. This class is discussed in detail in section 5, *Receiving Sequence Commands from Gemini*.
- GeminiUtil: This class provides methods that allow instrument code to interact with Gemini systems. In particular, it may be used to receive information from Gemini EPICS-based systems (primarily the TCS and

its subsystems), offload wavefront corrections to the Primary Control System (PCS) and access the TCS Context to compute World Coordinate Information. This class is detailed in section 6, *Interacting with Gemini Systems*.

- ServicesUtil: GIAPI provides a set of services that most of the instruments will need, such as logging, observatory time and GIAPI configuration properties. This class provides access to those services, and they are discussed in section 7, *Using Gemini Services*.
- DataUtil: The science data produced by the instruments must be transferred to Gemini where is stored, tracked, and delivered to observers and the Gemini Science Archive. The *DataUtil* class in the GIAPI C++ language glue provides the methods needed by instrument software create science datasets in the Gemini environment. Specifics about this class are discussed in section 8, *Providing Science Data to Gemini*.
- GiapiUtil: A collection of auxiliary methods available in the GIAPI to deal with error condition and fault tolerance mainly. Details about this class are discussed in section 9, Error Handling and Fault Tolerance in the GIAPI C++ Language Glue.

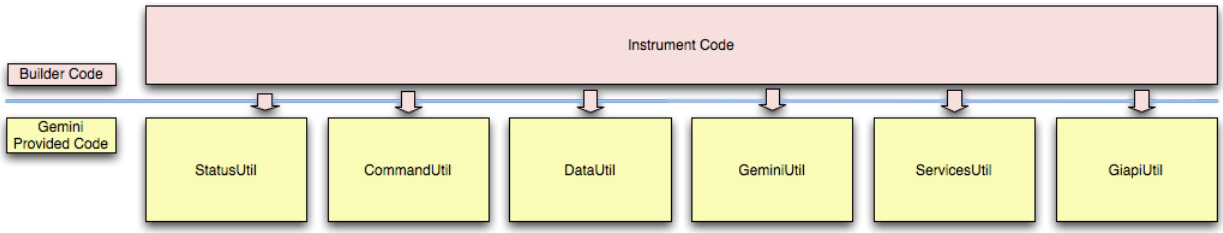


Figure 3: GIAPI C++ Language glue API as set of utility classes

The rest of this document is structured as follow. Section 3 presents a general description on the usage of the GIAPI C++ API, including library dependencies, tools and conventions used. The remaining sections cover each set of utility classes depicted above in turn.

3 Library Usage

This section discusses topics that are relevant to all GIAPI usage.

3.1 Compiler and build tools

The GIAPI C++ API was compiled with GCC version 4.1 on RedHat Enterprise Linux 5 Server for 64 bits. The following are tools used during the development of the GIAPI C++ API:

- g++ (GCC) 4.1.2 20080704 (Red Hat 4.1.2-46)
- GNU Make 3.81
- GNU ld version 2.17.50.0.6-2.el5 20061020
- RedHat Enterprise Linux 5 Server, Linux kernel 2.6.18-194.26.1.el5, 64-bit

The required compiler to build and link with the GIAPI C++ API is GCC version 4 or above. Version 3 of GCC does not include the Technical Report 1 extensions libraries used internally in the library so its usage is discouraged.

The dependencies described in the next section are known to work in this environment. A few other conventions used in this section are the following:

- The library is installed in a directory called `INSTALL_DIR`
- External dependencies are installed in a directory called `EXTERNAL_LIB`
- Library headers are in `INSTALL_DIR/include`, referred as `GIAPI_INCLUDE_DIR`
- Binary library is located in `INSTALL_DIR/lib`, referred as `GIAPI_LIB_DIR`

3.2 Dependencies

The GIAPI C++ API depends on the libraries listed in this section. Many of them should be readily available in the environment for which GIAPI is targeted. A few others must be installed and are prerequisites to use the C++ API. These cases are noted.

- **STL**: The Standard Template Library is used in the C++ API. The extensions available in the GNU C++ compiler declared in the namespace `__gnu_cxx` are used, but limited to the hash table implementations. Hash tables are not part of the STL, but the hash table implementation in the `__gnu_cxx` namespace is widely used. The STL and extensions should be available in the target environment. No extra libraries are included in the binary distribution of the GIAPI C++ Language glue. It is required they are available in the target system.

- **Smart Pointers**: The C++ Language glue implementation uses the `std::tr1::shared_ptr` library. This almost-standard smart pointer template provides non-intrusive reference-counting semantics for any class. It was originally designed and implemented as part of the *boost* libraries [8], but today is available in recent versions of the g++ and other compilers by default. Many of the boost libraries are part of the libraries already included in the C++ Standards Committee's Library Technical Report 1 (tr1). GNU C++ version 4 in Linux comes with the tr1 extensions already included. It is required that the `std::tr1::shared_ptr` library be available to build the C++ API.

- **The *any* boost library**: The *any* boost library is used internally in the GIAPI C++ Language glue. Details about this library can be found in http://www.boost.org/doc/html/any.html. This library is composed of C++ headers only that are included in the source code of the C++ API, so they are available directly. It is listed here for completeness.

- **The Apache Log4cxx library**: Log4cxx, a C++ port of Log4J, is used in the GIAPI C++ API. Details about how to use Log4cxx can be found in [7]. Log4cxx must be available both to compile and to use the GIAPI C++ API. Source code can be obtained via anonymous SVN from http://svn.apache.org/repos/asf/logging/log4cxx/trunk. During building, it is assumed it is installed in `EXTERNAL_LIB/log4cxx`. The binary distribution of the GIAPI C++ Language glue comes with the Log4cxx library available in the `GIAPI_LIB_DIR` directory.

- **CppUnit testing library**: For unit testing, the C++ API uses the CppUnit framework, described in [9]. This library is required to run the unit tests that come with the source code distribution. It is not required to use the library though. For building, it must be installed on `EXTERNAL_LIB/cppunit`. It is not included in the binary distribution of the GIAPI C++ API.

- **Apache ActiveMQ-CPP Library**: Communication between the GIAPI C++ API and the Gemini Master Process uses Java Message Service (JMS) [6]. ActiveMQ is the Message Broker provider that implements the JMS specification [10]. The JMS API for C++ for interfacing with Apache ActiveMQ is called C++ Messaging Service (CMS). This API is implemented by the Apache ActiveMQ CPP library [11]. This library is required to compile and use the GIAPI C++ API library. To build the GIAPI C++ library, ActiveMQ CPP must be installed in `EXTERNAL_LIB/activemq`. The binary distribution comes with this library in the `GIAPI_LIB_DIR` directory.

3.3 Accessing the library C++ public headers

Any necessary C++ headers to access the GIAPI C++ public interfaces are available in the `GIAPI_INCLUDE_DIR` directory. Usually it is only needed to add the `-I$(GIAPI_INCLUDE_DIR)` flag during compilation in order to have these C++ headers available to build the source code.

3.4 Linking with the library

The GIAPI C++ language glue is implemented as a shared library. The library is used by any process that the instrument builder creates that must communicate with Gemini. Instrument code that needs to interact with the GIAPI framework needs to link with this library.

The library is named **giapi-glue-cc**. The library is installed in the `INSTALL_DIR` directory. This directory contains two folders: `lib` and `include`. The `lib` directory contains the actual library shared object whereas the

include directory contains the C++ headers required to use the library. These two directories are referred to as the GIAPI_INCLUDE_DIR and the GIAPI_LIB_DIR respectively.

In Linux, the library file is named **libgiapi-glue-cc.so**. This is actually a symbolic link to the current version of the library that is in the same directory.

Since the GIAPI C++ API depends on external libraries (see section 3.2), to use the C++ API it is required to have those libraries accessible as well. The binary distribution of the GIAPI C++ API contains those libraries in the GIAPI_LIB_DIR.

To link against the installed libraries in a GIAPI_LIB_DIR directory under Linux, you must either use libtool, and specify the full pathname of the library, or use the -L$(GIAPI_LIB_DIR) flag during linking and do at least one of the following:

- add GIAPI_LIB_DIR to the LD_LIBRARY_PATH environment variable during execution.
- Add specified directory to library cache: (as root) ldconfig -n $(GIAPI_LIB_DIR). This will not permanently configure the system to include this directory. The information will be lost once upon system reboot.
- have your system administrator add GIAPI_LIB_DIR to /etc/ld.so.conf. See any operating system documentation about shared libraries for more information, such as the ld(1) and ld.so(8) manual pages.

To run an executable using the library, you need a configuration file called gmp.properties. This file should be either in your current directory, or you need to set the GMP_CONFIGURATION environment variable to point to it. You can find a sample gmp.properties file in the examples directory.

3.5 General conventions used in the C++ Language Glue API

The C++ API is implemented in the namespace *giapi*. In all the examples in this document, it is assumed that the `using namespace giapi` directive is in use. All the other namespaces used in the GIAPI C++ API are contained in the *giapi* namespace.

As mentioned in section 3.1, the C++ API uses shared pointers extensively through the std::tr1::shared_ptr library. For convenience, a type definition is provided in all the class headers that require access through a smart pointer. The convention is that for any given class Class, the corresponding smart pointer definition will be called pClass. For instance, the Configuration class (described in chapter 5) contains the following pConfiguration definition in the <giapi/Configuration.h> header file:

```
typedef std::tr1::shared_ptr<Configuration> pConfiguration
```

The <giapi/Configuration.h> header includes <tr1/memory> required to access the smart pointer library. In general, instrument code will only need to include the GIAPI C++ headers when using the smart pointer definitions in the GIAPI classes.

If the same smart pointer library is used outside the scope of classes provided in the GIAPI, then including <tr1/memory> is mandatory. In other words, instrument code should not rely on the inclusion of this header by the GIAPI headers for other purposes, since this is an implementation detail.

3.6 Exceptions in the C++ Language Glue API

Several methods in the GIAPI Language glue can produce exceptions that need to be handled by the client code. Details about the exception class hierarchy and the different types of exceptions in use in the GIAPI are found in section 9. The higher level methods in the GIAPI C++ Language glue abstract the complexity by listing only the base class of all the GIAPI exceptions, the giapi::GiapiException class. Methods that throw this exception are listed in the following sections.

4 Providing Status to Gemini

This section describes how an instrument provides status information to Gemini using the GIAPI C++ API discussed in section 9 of [2]. Please refer to this section for more information on status usage. We begin by providing an overview on how status is supported in the API and then a description of the different methods available in the library. Examples are shown to illustrate the principal functionalities.

4.1 General overview

The functionality required for providing status is provided in a utility class called `StatusUtil`. The three types of status items provided by the GIAPI are supported in the C++ API with the classes: `StatusItem`, `AlarmStatusItem` and `HealthStatusItem`. `StatusItem` is the base class of the other two.

In the GIAPI C++ API `StatusItem` contains the following information:

Status Item Member	Description	Example
name	The unique name of the status item.	INST:blockingfilter
timestamp[1]	The time when the status item value was set.	time is in milliseconds since January 1, 1970 UTC.
value	The current value of the status item.	10, 12.23, "Ready", K(prime)
data type	The type for values in the status item. All values must be of the same type.	Currently support for boolean, integer, float, double and string

1. Note that the timestamp is available in the status item, but it is set by the GIAPI as a side effect of setting the value of the item. The builder code does not need to set the timestamp.

Table 1: `StatusItem` members

The data types are defined in the `giapi::type` namespace, declared in the `<giapi/giapi.h>` header file. Supported types are `BOOLEAN`, `INT`, `FLOAT`, `DOUBLE` and `STRING`.

`AlarmStatusItem` adds a *severity*, a *cause* and a *message*. *Severity* and *cause* are defined as enumerated types in the GIAPI C++ API. The valid values for them are shown in the following table. These enumerated types are defined in the `giapi::alarm` namespace and are called `Severity` and `Cause`, respectively. They are also declared in the `<giapi/giapi.h>` header.

Status Item Member	Description	Values

severity	Severity indicates whether or not the alarm is catastrophic	`ALARM_OK`
		`ALARM_WARNING`
		`ALARM_FAILURE`

cause	Cause indicates the cause of an alarm.	`ALARM_CAUSE_OK`
		`ALARM_CAUSE_HIHI`
		`ALARM_CAUSE_HI`
		`ALARM_CAUSE_LOLO`
		`ALARM_CAUSE_LO`
		`ALARM_CAUSE_OTHER`

message	Message can be used to describe the cause of an	

	alarm condition. Message is required with the `ALARM_CAUSE_OTHER` alarm cause	

Table 2: Alarm status members. Enumerated types defined in the giapi::alarm namespace

Finally, `HealthStatusItems` are special status items whose valid values are one of the following enumerated types defined in the `giapi::health` namespace.

Health Value	Description
`GOOD`	The system/subsystem is normal
`WARNING`	The system/subsystem is operating, but not normally.
`BAD`	The system/subsystem is not operating.

Table 3: Health Values, defined in the giapi::health namespace

The enumerated values are defined in the `giapi::health` namespace, declared in the `<giapi/giapi.h>` header file. The full name of a health status value would be `giapi::health::GOOD` for instance.

All the status items need to be initialized in the GIAPI framework before using them. The initialization usually means to give them a name and an associated data type. For health status items the data type is fixed, therefore it is not needed during initialization.

Every time a value is set in a status item, the status item is flagged as *dirty*. A *dirty* status item means its value has not been published to the GMP. When the GIAPI C++ API posts the status items to the GMP, the status item *dirty* flags are cleared. If the items are not *dirty*, on the other hand, the items will not be posted since it is assumed their values are already known by the GMP. By default, the status items are initialized as *dirty*. The *dirty* flag is not set when the status item value is set with a value that is equal to the status item's current value.

Providing status to Gemini in the C++ API is done using the `StatusUtil` class. This class provides methods to handle all the required status item functionality.

The following table lists all the public classes and types that provide status support in the GIAPI C++ API and the C++ headers where they are declared. The `giapi` namespace is assumed in all the classes and types.

Class/Type	Description	C++ header
`StatusUtil`	Utility class to manipulate and post status to Gemini.	<giapi/StatusUtil.h>
`type::Type`	Enumerated type to define the supported data types for Status Items	<giapi/giapi.h>
`alarm::Severity`	Enumerated type to define the severity of alarms	<giapi/giapi.h>
`alarm::Cause`	Enumerated type to define the cause of alarms	<giapi/giapi.h>
`health::Health`	Enumerated type to define health status values	<giapi/giapi.h>

Table 4: Main classes and types to provide status support in the C++ Language glue

The set of tasks that a typical application must follow to use status within the GIAPI framework are easily identified. The following enumerates these common steps and gives pointer to sections in which they are covered in more detail:

1. Initialize status item (Section 4.2). Status items of the three types must be created before they can be used in the GIAPI. The initialization defines the type of the status item and usually the data type it contains (the exception to this is Health Status Items where the data type is pre-defined).
2. Set values to the status items (Section 4.3). Once initialized, status items are ready to be used to store different values. Every time a value changes in a status item, it is internally flagged as *dirty* and its timestamp is updated.
3. Post status items to the GMP (Section 4.4). When the application needs to publish its state, it will use the methods described in this section. During the post stage, the *dirty* items are sent to the GMP and their *dirty* state is cleared.

4.2 Status Item initialization

In order to use the different status items, they need to be initialized in the GIAPI framework. The following are the methods available in `StatusUtil` to do this.

createStatusItem

Signature:	`static int createStatusItem(const std::string& name, const type::Type type)`
Description:	Create an status item in the GIAPI framework
Parameters:	**name**: The name of the status item to be created **type**: Data type associated to the status item
Returns:	**status::OK** if the status item was successfully created **status::ERROR** if another status item with the same name already exists.

For instance:

```
if (StatusUtil::createStatusItem("inst:filter", type::STRING) != status::OK) {
    //status item couldn't be initialized
}
```

The above example is instantiating a status item “inst:filter”, that will store string elements on it.

createAlarmStatusItem

Signature:	`static int createAlarmStatusItem(const std::string& name, const type::Type type)`
Description:	Create an alarm status item in the GIAPI framework. An alarm status item is a status item that adds *Severity*, *Cause* and a *Message*. Default Severity is `alarm::ALARM_OK`. Default Cause is `alarm::ALARM_CAUSE_OK`. Default message is `NULL`
Parameters:	**name** The name of the alarm status item that will be created. If an status item with the same name already exists, the method will return **giapi::status::ERROR** **type** Type information for the values to be stored in the status item.
Returns:	**status::OK** if the alarm item was successfully created **status::ERROR** if another status item with the same name already exists.

For example, to create a temperature status item with alarm information, the following call could be used:

```
if (StatusUtil::createAlarmStatusItem("inst:dc:temperature", type::INT) != status::OK) {
    //alarm status item couldn't be initialized
}
```

createHealthStatusItem

Signature:	`static int createHealthStatusItem(const std::string& name)`
Description:	Create a health status item in the GIAPI framework. A health status item provides the overall operational status of a system or subsystem. The health can be: `health::GOOD`, `health::WARNING` or `health::BAD`. The default state for health after creation is `health::GOOD`.
Parameters:	**name** The name of the health status item that will be created. If an status item with the same name already exists, the method will return **giapi::status::ERROR**
Returns:	**status::OK** if the status item was successfully created **status::ERROR** if another status item with the same name already exists.

The following example will instantiate a health status item:

```
if (StatusUtil::createHealtStatusItem("inst:health") != status::OK) {
    //health status item couldn't be initialized
}
```

4.3 Setting Values to status items

Once initialized, the status items are ready to be used. The following sections describe the methods available in `StatusUtil` to set values to the different status items. Every time the value is changed, the internal state of the status item is marked as *dirty*. In the next *post* operation (see section 4.4), all the *dirty* status items will be dispatched to the GMP.

For normal status items the library provides methods to set values using different types. For alarm status items, the library allows setting or clearing the alarm state of the items. Finally, a method allows setting the health value to health status items. If an operation is not supported for a given status item, an error is returned by the call. For example, an error is returned if a status item is set with a value with a type different than the type used when the status item was initialized.

4.3.1 Setting values to regular status items

The following are the methods available in the C++ Language glue to set values to the different status items.

setValueAsInt

Signature:	`static int setValueAsInt(const std::string& name, int value)`
Description:	Set the value of the given status item to the provided integer value
Parameters:	**name:** Name of the status item whose value will be set **value:** Integer value to store in the status item
Returns:	**giapi::status::OK** if the value was set correctly **giapi::status::ERROR** if there is a problem setting the value and the operation was aborted. This can happen if the type of the status item was not defined as `type::INTEGER` or if there is no status item associated to the given name

To set the exposure time in the detector controller of an instrument, the following code might be used:

```
if (StatusUtil::setValueAsInt("inst:dc.exposure", 60) != status::OK) {
    //couldn't set the exposure time
}
```

In addition to the `setValueAsInt` method, the C++ Language Glue provides the following methods that operate on different data types:

- `setValueAsDouble`

- `setValueAsFloat`
- `setValueAsBoolean`
- `setValueAsString`
- `setValueAsFloat`

These methods have a similar signature to the `setValueAsInt` method so they are not described in more details in this document.

4.3.2 Setting Alarms

To set the alarm state of an `AlarmStatusItem`, the C++ API provides the following methods in the `StatusUtil` class:

setAlarm

Signature:	`static int setAlarm(const std::string& name,`
	`alarm::Severity severity,`
	`alarm::Cause cause,`
	`std::string& message = string());`

Description:	Set the alarm for the specified status alarm item.
Parameters:	**name** Name of the alarm item. The alarm items should have been initialized by a call to `createAlarmStatusItem`. Failing to do so will return an error. **severity** the alarm severity. **cause** the cause of the alarm **message** Optional message to describe the alarm. This argument is mandatory if the cause is `alarm::ALARM_CAUSE_OTHER`.
Returns:	**giapi::status::OK** if the alarm value was set correctly **giapi::status::ERROR** if there was an error setting the alarm (for instance, the alarm item has not been created or the name does not correspond to an alarm status item). This value is also returned if no message is specified when the cause is set to `alarm::ALARM_CAUSE_OTHER`.

For instance:

```
StatusUtil::setAlarm("inst:dc:temperature",
                     giapi::alarm::ALARM_WARNING,
                     giapi::alarm::ALARM_CAUSE_HI);
```

Note: Even though we are assuming in all the examples that we are using the namespace `giapi`, in the example above the `alarm` namespace is explicitly qualified with the `giapi` namespace to avoid conflicts with the alarm interface in the standard C library. This might not be necessary in some environments.

To clear the alarm state one option would be to use the call:

```
StatusUtil::setAlarm("inst:dc:temperature",
                     giapi::alarm::ALARM_OK,
                     giapi::alarm::ALARM_CAUSE_OK);
```

The C++ Language glue provides a convenience method to reset the state of an alarm. This is the recommended way to reset alarm states.

clearAlarm

Signature:	`static void clearAlarm(const std::string& name)`
Description:	Clear the alarm state of the alarm status item specified by name.

Parameters:	**name** Name of the alarm item. The alarm items should have been initialized by a call to `createAlarmStatusItem`. Failing to do so will return an error.
Returns:	**giapi::status::OK** if the alarm was cleared **giapi::status::ERROR** if there was an error clearing the alarm (for instance, the alarm has not been created, or the name is not associated to an alarm status item)

Using this method, to clear an alarm state will require only a call similar to:

```
StatusUtil::clearAlarm("inst:dc:temperature");
```

Note that if an alarm is to be used with a status item, it must be initialized as an `AlarmStatusItem`; a `StatusItem` will not allow the alarm to be set.

4.3.3 Setting Health

For health status items, the C++ Language glue provides one method in the `StatusUtil` class:

setHealth

Signature:	`static int setHealth(const std::string& name,` `const health::Health health)`
Description:	Set the health value for the given health status item.
Parameters:	**name** Name of the health item. The health item must have been initialized by a call to `createHealthStatusItem()`. Failing to do so will return an error. **health** the health state of the health status item specified by name
Returns:	**giapi::status::OK** if the health was successfully set **giapi::status::ERROR** if there was an error setting the health (for instance, the health status item hasn't been created or the name doesn't correspond to a health status item).

As an example, the following method will set the health to `WARNING` to the status item **inst:health**. The assumption here is that **inst:health** was previously initialized by the `createHealthStatusItem` call.

```
if (StatusUtil::setHealth("inst:health", health::WARNING) != status::OK) {
    //problem setting health
}
```

4.4 Posting status to Gemini

Any application that needs to publish its status to Gemini must create and keep up to date different status items. For these, instrument use the methods described in section 4.2 and 4.3. To publish the values of these status items with the GMP to Gemini, instruments need to post their status items frequently following the discussion in Section 9 of [2]. The *post* mechanisms allow the application to provide updates to Gemini through the GMP.

In the C++ Language glue, the `StatusUtil` class provides methods to help posting status to Gemini. These methods are described now.

postStatus

Signature:	`static int postStatus() throw (GiapiException)`
Description:	Post all pending status to Gemini. Pending statuses are those whose value has changed since last time posted.
Parameters:	**none**
Returns:	**giapi::status::OK** if the post succeeded. **giapi::status::ERROR** if there is some error in the attempt to send

Throws:	**GiapiException** in case there is a problem with the underlying mechanisms to execute the post

For instance, the following call will post all the items that have changed since the last time the same call was executed.

```
try {
    if (StatusUtil::postStatus() != giapi::status::OK) {
        //an error occurred.
    }
} catch (GiapiException& e) {
    //handle exception here.

}
```

In addition, the C++ language glue allows instrument code to post one specific item to the GMP, using a `postStatus` method with the following signature:

postStatus

Signature:	`static int postStatus(const std::string& name) throw (GiapiException)`
Description:	Post the specified status item to Gemini. The status will be sent only if it has changed since the last time it was posted.
Parameters:	**name** The name of the status item to be posted
Returns:	**giapi::status::OK** if the post succeeds. . **giapi::status::ERROR** if there is no information associated to the specified status item or the item was not posted since it did not change since the last time it was posted to the GMP
Throws:	**GiapiException** in case there is a problem with the underlying mechanisms to execute the post

The usage of this method is similar to the `postStatus()` method without argument.

5 Receiving Sequence Commands from Gemini

This section describes how an instrument receives and handles sequence commands from Gemini using the GIAPI C++ Language Glue. It provides the C++ language implementation for Section 10 of [2].

5.1 General Overview

For the builder, there are two parts to handling commands through GIAPI. First, the instrument registers interest in receiving a command using a sequence command handler. Then, through other GIAPI methods, the instrument provides command completion information to GMP and Gemini.

In the C++ API, the `CommandUtil` utility class provides the main access point to the functionality needed to receive and process sequence commands. Instrument code needs to provide implementations of the `SequenceCommandHandler` interface to define handlers that will be invoked when sequence commands are received.

A `SequenceCommandHandler` is registered to handle a given `SequenceCommand` and a set of `Activity` elements. The C++ API uses an `ActivitySet` enumerated type to represent groups of `Activity` items. One `SequenceCommandHandler` can be used to handle several `SequenceCommands` or each `SequenceCommand` can have its own.

A `SequenceCommandHandler` receives a `Configuration` associated to a given `SequenceCommand` and `Activity`. A `Configuration` is made up of parameters. Each parameter has a name and a value. In the C++ API, `Configuration` is an interface providing mechanisms to access the parameters contained in the configuration.

Whenever a `SequenceHandler` is invoked, a `HandlerResponse` is returned. In addition, a `HandlerResponse` is also used to provide command completion information to Gemini. Details about the use of the `HandlerResponse` are discussed further in the following sections.

The following table lists all the public classes that provide command support in the C++ language glue, and the C++ headers where they are declared.

Class	Description	C++ header
`CommandUtil`	Utility class to register interest in receiving commands and posting completion information.	<giapi/CommandUtil.h>
`SequenceCommandHandler`	Interface for Sequence Command Handlers	<giapi/SequenceCommandHandler.h>
`HandlerResponse`	Contains the response information when dealing with Commands	<giapi/HandlerResponse.h>
`Configuration`	Interface to access sequence command configurations	<giapi/Configuration.h>

Table 5: Main classes to receive commands in the C++ Language glue

The `SequenceCommand` in the GIAPI C++ Language Glue is defined as an enumerated type, listed in Table 6. The enumeration is called `giapi::command::SequenceCommand`. Activities are also defined as an enumerated type in the same namespace, called `giapi::command::Activity`. The values for Activities are listed in Table 7. In addition, the C++ API provides a `giapi::command::ActivitySet` enumerated type to help grouping Activities, whose values are in Table 8. Further discussion of `ActivitySet` is presented in section 5.3.

Sequence Command	Enum Value
test	`TEST`
reboot	`REBOOT`
init	`INIT`
datum	`DATUM`
park	`PARK`
verify	`VERIFY`
endVerify	`END_VERIFY`
guide	`GUIDE`
endGuide	`END_GUIDE`

Sequence Command	Enum Value
apply	`APPLY`
observe	`OBSERVE`
endObserve	`END_OBSERVE`
pause	`PAUSE`
continue	`CONTINUE`
stop	`STOP`
abort	`ABORT`
engineering	`ENGINEERING`

Table 6: Sequence Command enumeration values.

Activity Name	Enum Value	Description
preset	`PRESET`	Receiver should only determine if the actions associated with the sequence command can be started and return a handler response.
start	`START`	Receiver should start any actions associated with the sequence command and return a handler response.
preset*/*start	`PRESET_START`	Receiver should execute ***preset***, and if successful

		immediately start the actions associated with the sequence command. The receiver should return the appropriate handler response.
cancel	`CANCEL`	Receiver should attempt to cancel the actions associated with the request ID and sequence command.

Table 7: Enumeration values of Activities and descriptions

ActivitySet Enum Value	Description
`SET_PRESET`	A set containing the `PRESET` activity only
`SET_START`	A set containing the `START` activity only
`SET_PRESET_START`	A set containing the `PRESET_START` activity only
`SET_CANCEL`	A set containing the `CANCEL` activity only
`SET_PRESET_CANCEL`	A set containing the `PRESET` and `CANCEL` activities
`SET_START_CANCEL`	A set containing the `START` and `CANCEL` activities
`SET_PRESET_START_CANCEL`	A set containing the `PRESET`, `START` and `CANCEL` activities.

Table 8: ActivitySet enumeration values

These three enumerated types are defined in the giapi::command namespace declared in the <giapi/giapi.h> header file; therefore the fully qualified name of one enumerated type is `giapi::command::REBOOT`, for instance.

The tasks that a typical application must follow to receive commands from Gemini are listed below, including pointer to sections in which they are covered in more detail:

1. Implement a sequence command handler (Section 5.2)
2. Subscribe to a sequence command (Section 5.3 and 5.4)
3. Post completion information to Gemini (Section 5.5)

5.2 Sequence Command Handlers

A Sequence Command Handler (SCH) in the C++ API is an instance of a class that implements the `SequenceCommandHandler` interface. One SCH can be used for multiple sequence commands or each sequence command can have its own. The interface defines only one method named `handle`. The following is the signature of the `handle` method in the C++ Language glue:

handle

Signature:	`pHandlerResponse handle(command::ActionId id,`
	`command::SequenceCommand sequenceCommand,`
	`command::Activity activity,`
	`pConfiguration config)`

Description:	This method is invoked when commands are received by the GIAPI if the handler has been registered using the *subscribeSequenceCommand* or the *subscribeApplyCommand* methods.

Parameters:	**id** the unique value that identifies the request or equivalently identifies the set of actions started by the request. The handler should retain the value unless the actions started by the request are completed immediately. The same action id will not be used again in any future request unless the GMP is restarted. **sequenceCommand** the sequence command that should be handled, such as TEST or REBOOT **activity** Activity requested by the sender, like PRESET_START or CANCEL **config** a configuration associated to the command, as a smart pointer.
Returns:	A smart pointer to a HandlerResponse object, containing the result of handling the command.

The possible values for the `sequenceCommand` and the `activity` arguments are shown in Table 6 and Table 7. The `Configuration` is an interface passed to the handler as a smart pointer. There are three principal operations provided by the `Configuration` interface:

1. Get the value associated to any given key
2. Get all the keys (names) of the parameters contained in the configuration
3. Get the number of parameters contained in the configuration

The methods that provide support for these operations are described below. They are part of the `Configuration` interface. Notice the handle method receives as an argument a *smart pointer* to the actual `Configuration`, so the client code does not need to worry about destroying this object.

10

getValue

Signature:	`const std::string getValue(const std::string & key)`
Description:	Return the value associated to the given key. The key is the name of the parameter associated to the value, like inst:cc.filterA. If the key is not present, it returns `NULL`
Parameters:	**key** the name of the parameter whose value we need to retrieve
Returns:	The value associated to the given key or `NULL` if there is no value associated for the key in the current configuration.

getKeys

Signature:	`vector<std::string> getKeys() const`
Description:	Return the keys contained in the configuration. If the configuration does not contain any key, an empty vector is returned.
Parameters:	**none**
Returns:	A vector of strings representing the keys contained in the configuration. An empty list is returned if no keys are present.

getSize

Signature:	`int getSize() const`
Description:	Return the number of parameters contained in this configuration.
Parameters:	**none**
Returns:	The total number of parameters contained in this configuration.

A smart pointer to a `HandlerResponse` is returned whenever the handle method is invoked. `HandlerResponse` is an object that contains an enumerated response type and a string message when an error is produced. The `HandlerResponse` is used to indicate the end of the ACM acceptance phase.

Table 9 lists the different response types provided in the C++ API. The enumerated type is called `Response`. The values are defined in the `HandlerResponse` class, so a fully qualified enumerated value would be `giapi::HandlerResponse::ACCEPTED`, for instance. Since implementers of the `HandlerResponse` interface must include the `<giapi/HandlerResponse.h>` header file, there is no need for additional headers.

Handler Response Name	Enum Value	Description
preset accepted	`ACCEPTED`	Receiver evaluated the request. Returns ACCEPTED when the actions can be started immediately. This is the success response for the ***preset*** activity.
actions started	`STARTED`	Receiver evaluated the request (preset) and started the actions successfully. This is the one success response for the ***start*** and ***preset/start*** activity (the other being `COMPLETED`).
actions completed	`COMPLETED`	Receiver evaluated the request (preset) started and completed the actions. This is reserved for short commands that don't move devices. This is one success response for the ***start*** and ***preset/start*** activity (the other being `STARTED`).
error	`ERROR`	Request ended with an error. All activities can return `ERROR`.

Table 9: `Response` types and descriptions

In order to simplify the memory management of newly instantiated `HandlerResponse` objects, the GIAPI encapsulates the use of these objects in smart pointers wrappers. The handle method returns a smart pointer containing the `HandlerResponse` object. This smart pointer is referred as a `pHandlerResponse` in the GIAPI context, which is just type definition to the appropriate data structures (see section 3.5 for details about this convention). The `HandlerResponse` class simplifies this process by providing two factory methods to initialize smart pointers.

The first factory method takes as an argument the `Response` type to construct the instance. Here is the signature of that method:

create

Signature:	`static pHandlerResponse create(const Response type);`
Description:	Static factory for a `HandlerResponse` of the given Response `type`
Parameters:	**type** The Response type of the HandlerResponse to be instantiated.
Returns:	A `pHandlerResponse`, a smart pointer referring to a `HandlerResponse` object. Since these objects cannot be modified by the user (immutable), this call can return the same instance when the same argument is used.

Here is an example of how the handler code could look like:

```
pHandlerResponse handle(command::ActionId id,
                        command::SequenceCommand sequenceCommand,
                        command::Activity activity,
                        pConfiguration config) {
    //handle the sequence command
    …
    //construct handler response. Notice the use of the factory method
    pHandlerResponse response = HandlerResponse::create(HandlerResponse::ACCEPTED);
```

```
    return response;
}
```

The above example illustrates the first factory method in the `HandlerResponse`, Additionally, the C++ Language glue provides another factory constructor that takes as an argument a string and will instantiate an error handler response with a given message. The signature of that method is as follows:

create

Signature:	`static pHandlerResponse createError(const std::string & msg);`
Description:	Static factory for an error `HandlerResponse`. The response type is set to `ERROR` and the error message is stored.
Parameters:	**msg** The error message to be used for this error `HandlerResponse`.
Returns:	A smart pointer to a new allocated error `HandlerResponse` object.

Since the `HandlerResponse` interface does not provide methods to modify an instance of the class, this last factory method is the mechanism to instantiate *error* `HandlerResponse` objects. An attempt to initialize an *error* `HandlerResponse` using the `HandlerResponse.create()` method is a mistake, since the error message will not get initialized.

In order to get the error message associated to an `ERROR HandlerResponse`, the `getMessage()` method is provided in the `HandlerResponse` class:

getMessage

Signature:	`std::string getMessage() const`
Description:	Return the message associated to this handler response. If the handler is not `ERROR`, the returned string will be an empty string.
Parameters:	**None**

5.3 Subscribing to Sequence Commands other than Apply

The previous section presented the Sequence Command Handler. This section shows how to associate an SCH with a sequence command. This section discusses sequence command other than **apply**. See section 10.7 of [2] for more information about subscribing to other sequence commands.

It is a requirement that a Sequence Command Handler is registered for each one of the sequence commands (See section 10.7 of [2]). If a particular Sequence Command is of no interest to an instrument, a Sequence Command Handler that returns immediately an `ACCEPTED HandlerResponse` should be provided.

To subscribe to a sequence command with a handler, the `subscribeSequenceCommand` method in the `CommandUtil` class is used.

subscribeSequenceCommand

Signature:	`static int subscribeSequenceCommand(command::SequenceCommand id`
	`command::ActivitySet activities,`
	`pSequenceCommandHandler handler) throw (GiapiException)`

Description:	Associates the given **handler** to the sequence command specified by the **id** argument. The **activities** argument allows different handlers for different activities. For instance, one handler can implement only `PRESET` and another `START`.
Parameters:	**id** The SequenceCommand enumerated type that identifies the sequence command to which a handler will be associated.

	activities Enumerated type representing the set of Activities the handler will be associated with for the given prefix. **handler**. Pointer to a concrete implementation of a SequenceCommandHandler that will be associated to the given sequence command and actions. The most recently registered sequence command handler for a sequence command will be used.
Returns:	**gapi::status::OK** if the subscription succeeds. Otherwise, it returns **giapi::status::ERROR**.
Throws:	**GiapiException** in case there is a problem with the underlying mechanisms to execute the subscribe operation

A few notes regarding the `subscribeSequenceCommand` method

- `command::SequenceCommand` is the name of the type for the enumerated values in Table 6
- The second argument, `command::ActivitySet` is the name of the type for the enumerated values from Table 7. This is to support using different handlers for different activities.
- The handler is passed as a smart reference pointer. The `pSequenceCommandHandler` type is defined in the `<giapi/SequenceCommandHandler.h>` header file.

To simplify the instantiation of `SequenceCommandHandler` instances and to avoid potential resource leaks, the following pattern is suggested when using a handler called `MyHandler`.

1. Never ever type `MyHandler *` in the client code. In other words, do not declare nor use direct pointers to the handler.
2. Define the constructor(s) for the sequence command handler as private, to prevent direct instantiation of new handlers in the client code
3. Provide a static factory to instantiate the handler, returning `pSequenceCommandHandler` objects.

The following code snippet illustrates this pattern. It defines a `SequenceCommanHandler` called `MyHandler`, providing a static factory to instantiate it.

```
class MyHandler: public SequenceCommandHandler {
public:
    virtual pHandlerResponse handle(command::ActionId id,
                            command::SequenceCommand sequenceCommand,
                            command::Activity activity, pConfiguration config) {
        pHandlerResponse response;
        //Handle the command, set the response
        ...
        return response;
    }
    static pSequenceCommandHandler create() {
        pSequenceCommandHandler instance(new MyHandler());
        return instance;
    }
private:
    MyHandler() {}
};
```

Notice that with this design, code like `MyHandler * handler = new MyHandler()` (direct pointer manipulation) is prohibited at compile time. In order to instantiate `MyHandler` objects and register them with a sequence command, the client code will need to use the factory provided, as in the following example:

```
pSequenceCommandHandler handler = MyHandler::create();
CommandUtil::subscribeSequenceCommand(command::REBOOT,
                                      command::SET_PRESET_START, handler);
```

5.3.1 Arguments in Sequence Commands

The **observe, reboot** and **engineering** sequence commands (in addition to **apply**) take arguments. The arguments for a sequence command are in the `Configuration` that is passed to the `handle` method in the `SequenceCommandHandler`. To read a specific argument, the instrument must know the keyword for the argument.

The keywords are defined as static constant strings in the `Configuration` class. The following table lists the argument keywords and the sequence commands that use them:

Argument Keyword	Sequence Command	Description	Values
DATA_LABEL	`OBSERVE`	The data label the observe sequence command will produce. The argument should be used to create the file for the dataset in the form datalabel.fits.	String value
REBOOT_OPT	`REBOOT`	Used within the GIAPI on Linux. The argument allows specifying different actions the instrument should take when the reboot command is received.	REBOOT, GMP, NONE
COMMAND_NAME	`ENGINEERING`	Used by the instrument to determine which of its engineering commands to execute.	String value
…	`ENGINEERING`	The engineering sequence command can take any number of optional parameters. They will be passed to the instrument to be used as parameters for the selected engineering command. If the parameters needed are of a different type, it is the instrument software's responsibility to parse the string and do the conversion.	String value

Table 10: Keywords to identify arguments in sequence commands

The valid values for the **reboot** sequence command (REBOOT, GMP, NONE) are also declared in the `Configuration` class, for convenience.

The following code snippet illustrates the handle method of an **observe** sequence command handler. This handler gets the data label from the configuration and stores it in a class member variable for later use.

```
pHandlerResponse handle(command::ActionId id,
                        command::SequenceCommand sequenceCommand,
                        command::Activity activity, pConfiguration config) {

    if (sequenceCommand == command::OBSERVE) {
        //Get the data label argument in the observe command
        _datalabel = config->getValue(Configuration::DATA_LABEL);
```

```
    }
    pHandlerResponse response;
    //prepare response
    ...
    return response;
}
```

5.4 Subscribing to the Apply Sequence Command

Handling the **apply** command is potentially more complex. A configuration passed to an instrument from the OCS may have parts that are handled by different components within the instrument implying different handlers.

To support this, the C++ Language glue provides the `subscribeApply` method in the `CommandUtil` class. The following is the description of this method:

subscribeApply

Signature:	`static int subscribeApply(const std::string & prefix,`
	`command::ActivitySet activities,`
	`pSequenceCommandHandler handler) throw (GiapiException)`

Description:	Associates the given handler to the configuration prefix specified and array of Activity elements. When the system receives a configuration it will split it up according to the registered handlers and pass each the part of the configuration that is registered to handle. The **ActivitySet** argument allows different handlers for different activities. For instance, one handler can implement only `PRESET` and another `START`.
Parameters:	**prefix** the configuration prefix this handler will take care of. When a configuration is received by the system, it will be split and the part indicated by this prefix will be passed to the handler. **activities** Enumerated type representing the set of Activities the handler will be associated with for the given prefix. **handler**. Pointer to a concrete implementation of a SequenceCommandHandler that will be associated to the given sequence command and actions. The most recently registered sequence command handler for a sequence command will be used.
Returns:	**gapi::status::OK** if the subscription succeeds. Otherwise, it returns **giapi::status::ERROR**.
Throws:	**GiapiException** in case there is a problem with the underlying mechanisms to execute the subscribe operation

The following example code shows how to split the handler of configurations between a handler for DC and a handler for CC:

```
CommandUtil::subscribeApply("NICI:CC", giapi::command::SET_PRESET_START, handler1);
CommandUtil::subscribeApply("NICI:DC", giapi::command::SET_PRESET_START, handler2);
```

In this case the ***preset*** and ***start*** activities for NICI:CC will be handled by handler1 while ***preset*** and ***start*** for NICI:DC will be handled by handler2. The handler1 and handler2 are `SequenceCommandHandler` smart pointers, and should have been instantiated using a similar mechanism to what described in section 5.3.

5.5 Updating Gemini with Action Status

The last concept needed for handling commands is the posting of completion information to the GMP for actions that do not complete immediately. This case is triggered when the `SequenceCommandHandler` returns `STARTED` in the `HandlerResponse`. This case will occur most frequently in the CC when hardware devices must be moved to match the requested configuration. In this case, the SCH checks that it can start the actions, starts them and returns a `STARTED` response. Later, when the actions complete—either successfully or with an

error—some software component must notify the GMP of the completion status of the actions associated with the original `ActionID`.

The method call in the GIAPI to send completion information is provided in the `CommandUtil` class, and has the following signature:

postCompletionInfo

Signature:	`static int postCompletionInfo(command::ActionId id, pHandlerResponse response) throw (GiapiException)`
Description:	Post completion information to the GMP for actions that do not complete immediately. This case is triggered when a **SequenceCommandHandler** returns `STARTED` in the **HandlerResponse**. When the actions complete successfully or with an error, the system will notify the GMP of the completion status of the actions associated with the original **ActionId** using this call.
Parameters:	**id** the original **ActionId** associated to the actions for which we are reporting completion info. **response** contains the completion state associated to the **ActionId**. Valid response type are only `COMPLETED` and `ERROR`. In case the response is `ERROR`, a message should be provided in the **HandlerResponse** object
Returns:	**gapi::status::OK** if the post succeeds. Otherwise, it returns **giapi::status::ERROR**. **giapi:status:ERROR** is returned also if the response is not `COMPLETED` or `ERROR`, or if the response is `ERROR` and no message is provided.
Throws:	**GiapiException** in case there is a problem with the underlying mechanisms to execute the post operation

In a very simple application, the `handle` method is invoked for a given registered sequence command handler. The `ActionId` is stored:

```
pHandlerResponse handle(command::ActionId id,
                        command::SequenceCommand sequenceCommand,
                        command::Activity activity,
                        pConfiguration config) {
    //stores the action id.
    _id = id;
    //handling code here, most likely in a different thread
    ...
    //construct a handler response, in this case assuming actions started.
    pHandlerResponse response = HandlerResponse::create(HandlerResponse::STARTED);
    return response;
}
```

Later, when the instrument code wants to send completion info from the execution thread that started the actions, it will use the stored `ActionId` in the `postCompletionInfo` :

```
//construct a handler response with either ERROR or COMPLETED
pHandlerResponse response = HandlerResponse::create(HandlerResponse::COMPLETED);
//send completion info
CommandUtil::postCompletionInfo(_id, response);
```

Note this is a simplified example used only for illustration. In a real-life application the instrument code should report completion info using the `postCompletionInfo` only when all the handler's ongoing actions are completed and always using the most recent `ActionID`. Completion of the most recent `ActionID` means that actions with all other previous `ActionIDs` are also completed. See section 10.9 of [2] for a complete discussion of command completion.

6 Interacting with Gemini Systems

This section describes how an instrument interacts with other Gemini Systems using the GIAPI C++ Language Glue. This section parallels section 11 of [2].

6.1 General Overview

GIAPI provides methods for instrument code to interact with Gemini Systems in a controlled way. The following lists the supported operations currently available through the C++ Language Glue:

1. Monitor information from Gemini EPICS Channels
2. Provide wavefront sensor corrections to the PCS
3. Provide access to the TCS context to produce WCS information

Monitoring a Gemini EPICS channel requires registering an implementation of the `EpicsStatusHandler` interface. The interface defines a `channelChanged` method, which will be invoked every time the EPICS channel is updated. The argument of this method is an `EpicsStatusItem` that provides access to the EPICS channel data. Specifics of this functionality are described in section 6.2

Wavefront sensor corrections to the PCS are supported through a single call in the GIAPI C++ Language glue, `GeminiUtil::postPCSUpdate`, which takes an array of doubles and passes them to the PCS. This method is described in section 6.4.

The TCS context is represented in the GIAPI C++ Language glue as a structure called `TCSContext`. To get an update, a reference to this structure is passed to the `GeminiUtil::getTCSContext` method in the C++ Language glue. Details about the `TCSContext` structure and this method are described in section 6.5.

The methods to interact with Gemini Systems are provided by the `GeminiUtil` utility class. The following are the main classes the instrument code will need to work with to access these services and the C++ headers where they are defined.

Class	Description	C++ header
`GeminiUtil`	Utility class acting as a façade to the different Gemini Systems.	<giapi/GeminiUtil.h>
`EpicsStatusHandler`	Interface for EPICS status handlers	<giapi/EpicsStatusHandler.h>
`EpicsStatusItem`	Interface to access the content of a monitored EPICS channel.	<giapi/EpicsStatusItem.h>
`TCSContext`	C++ structure with the information of the TCS context	<giapi/giapi.h>

Table 11: Main classes to provide interaction with Gemini Systems in the C++ Language glue

6.2 Receiving Status Information from a Gemini System

In this use case, the instrument needs to subscribe to a status item that is published by one of the EPICS-based systems—typically the TCS or one of its subsystems.

Instrument code needs to provide an implementation of the `EpicsStatusHandler` interface. The interface defines only one method named `channelChanged`. The following is the signature of the `channelChanged` method in the C++ Language glue

channelChanged

Signature:	`void channelChanged(pEpicsStatusItem item)`
Description:	Callback invoked when the EPICS status is updated. Instrument code will use the `item` in this method

	to access the monitored EPICS status item data
Parameters:	**item** A smart pointer to the EPICS status item being monitored.
Returns:	**void**

The `pEpicsStatusItem` is a smart pointer definition to an `EpicsStatusItem` interface. This interface allows the handler to access the data contained in the monitored channel. The interface provides the following methods:

getName

Signature:	`const std::string getName() const`
Description:	Return the name of the EPICS channel represented by this status item
Parameters:	**none**
Returns:	Name of the EPICS channel represented by this EPICS status item

getType

Signature:	`type::Type getType() const`
Description:	Return the data type of the values contained in this EPICS status item
Parameters:	**none**
Returns:	The data type of the value or values contained in this EPICS status item. Types are defined under the giapi::type namespace in the <giapi/giapi.h> header file

getCount

Signature:	`int getCount() const`
Description:	Return the number of elements stored in the data part of this epics status item. All the elements are of the same data type, as returned by the `getType` method.
Parameters:	**none**
Returns:	Number of elements stored in the data part of this EPICS status element

getData

Signature:	`const void * getData() const`
Description:	Provides access to the data associated to the EPICS status item.
Parameters:	**none**
Returns:	Pointer to the data contained in this EPICS status item.

By using the `getType` and `getCount` methods, the client code can determine how to interpret the data contained in the data section of the `EpicsStatusItem`.

The following are auxiliary methods available in the `EpicsStatusItem` interface to help decoding the data.

getDataAsString

Signature:	`const std::string getDataAsString(int index) const throw (InvalidOperation)`

Description:	Read the data element at the specified index position as a string
Parameters:	**index** index of the element to retrieve. The index must be in the range 0 <= index < `getCount()`
Returns:	Data element in the specified index as a standard string.
Throws:	**InvalidOperationException** if the status item type is not a string or the index provided is greater or equals to the number of elements in this status item (as returned by the `getCount()` method)

getDataAsInt

Signature:	`int getDataAsInt(int index) const throw (InvalidOperation)`
Description:	Read the data element at the specified index position as an integer
Parameters:	**index** index of the element to retrieve. The index must be in the range 0 <= index < `getCount()`
Returns:	Data element in the specified index as an integer value.
Throws:	**InvalidOperationException** if the status item type is not an integer or the index provided is greater or equals to the number of elements in this status item (as returned by the `getCount()` method)

getDataAsFloat

Signature:	`float getDataAsFloat(int index) const throw (InvalidOperation)`
Description:	Read the data element at the specified index position as a float value
Parameters:	**index** index of the element to retrieve. The index must be in the range 0 <= index < `getCount()`
Returns:	Data element in the specified index as a float value.
Throws:	**InvalidOperationException** if the status item type is not a float or the index provided is greater or equals to the number of elements in this status item (as returned by the `getCount()` method)

getDataAsDouble

Signature:	`double getDataAsDouble(int index) const throw (InvalidOperation)`
Description:	Read the data element at the specified index position as a double value
Parameters:	**index** index of the element to retrieve. The index must be in the range 0 <= index < `getCount()`
Returns:	Data element in the specified index as a double value.
Throws:	**InvalidOperationException** if the status item type is not a double or the index provided is greater or equals to the number of elements in this status item (as returned by the `getCount()` method)

getDataAsByte

Signature:	`unsigned char getDataAsByte(int index) const throw (InvalidOperation)`
Description:	Read the data element at the specified index position as a byte value
Parameters:	**index** index of the element to retrieve. The index must be in the range 0 <= index < `getCount()`
Returns:	Data element in the specified index as a byte value.
Throws:	**InvalidOperationException** if the status item type is not a byte or the index provided is greater or

| equals to the number of elements in this status item (as returned by the `getCount()` method)

Once the `EpicsStatusHandler` is defined, it can be subscribed to or unsubscribed from an actual EPICS channel using methods available in the `GeminiUtil` class. To register a handler, the following method is used:

subscribeEpicsStatus

Signature:	`static int subscribeEpicsStatus(const std::string & name, EpicsStatusHandler handler) throw (GiapiException)`
Description:	Register a handler to receive updates when the specified EPICS status item is updated. This method is called with the `name` of the EPICS status item and a `handler` that will be called when the EPICS channel publishes an update.
Parameters:	**name** Name of the EPICS status item that will be monitored. **handler** Handler that will be called when an update is published. The most recently registered EPICS status handler will be used.
Returns:	**gapi::status::OK** if the subscription was successful. Otherwise, it returns **giapi::status::ERROR**.
Throws:	**GiapiException** in case there is an error accessing the GMP to register to receive updates for the given epics status item

In order to unregister a handler, the following method is available:

unsubscribeEpicsStatus

Signature:	`static int unsubscribeEpicsStatus(const std::string & name) throw (GiapiException)`
Description:	Unregister any handlers that might be associated to the given Epics status item.
Parameters:	**name** Name of the Epics status item that no longer will be monitored.
Returns:	**gapi::status::OK** if the deregistration was successful. Otherwise, it returns **giapi::status::ERROR**.
Throws:	**GiapiException** in case there is an error accessing the GMP to unregister to receive updates for the given epics status item

Note the `subscribeEpicsStatus` method receives as an argument a smart pointer to an `EpicssStatusHandler`. The `pEpicsStatusHandler` is defined in the `<giapi/EpicsStatusHandler.h>`. To avoid potential resource leaks during the instantiation of `EpicsStatusHandler` instances, the following pattern is suggested to implement an `EpicsStatusHandler` called `MyHandler`.

1. Never ever type `MyHandler *` in the client code. In other words, do not declare nor use direct pointers to the handler.
2. Define private constructor(s) for the `EpicsStatusHandler`, to prevent direct instantiation of new handlers in the client code
3. Provide a static factory to instantiate the handler, returning `pEpicsStatusHandler` objects, a smart pointer object that will refer to an actual `EpicsStatusHandler` instance.

A simple example will help to illustrate this idea. The following code snippet defines an `EpicsStatusHandler` called `MyHandler`, providing a static factory to instantiate it.

```
class MyHandler: public EpicsStatusHandler {
```

```
public:
    virtual void channelChanged(pEpicsStatusItem item) {
        //do something with the item
    }
    static pEpicsStatusHandler create() {
        pEpicsStatusHandler instance = pEpicsStatusHandler(new MyHandler());
        return instance;
    }
 private:
    MyHandler() {}
};
```

Using this pattern, an instrument that needs to be kept up to date with the value of the telescope altitude can use the following code:

```
pEpicsStatusHandler handler = MyHandler::create();
GeminiUtil::subscribeEpicsStatus("tcs:sad:currentEl", handler);
```

Notice that with this design, code like `MyHandler * handler = new MyHandler()` (direct pointer manipulation) is prohibited at compile time.

In this example, the method `channelChanged` will be called as the current altitude changes with an updated `pEpicsStatusItem` that will allow the handler code to read the altitude value.

6.3 Access to last known EPICS values

The GIAPI provides a `getChannel` method that can get the latest known value of a specific channel. Unlike the subcription mechanism described in section 6.2, the method will provide only the current value rather than updates. This can be useful e.g. in cases where a channel does not change very often.

The `getChannel` method has the following signature:

getChannel

Signature:	`pEpicsStatusItem getChannel(const std::string & name, long timeout) throw (GiapiException)`
Description:	Called to retrieve an EPICS status supported by the GMP. Instrument code can use the return value to access the latest EPICS status item data
Parameters:	**name** The name of the requested channel **timeout** Time to wait for a response in milliseconds
Throws:	**GiapiException** in case there is an error accessing the GMP or the requested EPICS channel
Returns:	`pEpicsStatusItem`

The `pEpicsStatusItem` is a smart pointer definition to an `EpicsStatusItem` interface as described in section 6.2

6.4 Provide Wavefront Sensor corrections to the PCS

Some instruments with on-board wave front sensors need to provide wavefront corrections to the Primary Control System (PCS). These wavefront updates are in the form of a slowly changing set of Zernike coefficients.

The GIAPI provides a `postPcsUpdate` method the instrument can use to update the PCS through the TCS. This method is available in the `GeminiUtil` class and has the following signature:

postPCSUpdate

Signature:	`static int postPcsUpdate(double zernikes[], int size) throw (GiapiException)`
Description:	Offload wavefront corrections to the Primary Control System (PCS). These wavefront sensor updates

	are in the form of slowly changing set of zernike coefficients.
Parameters:	**zernikes** array of zernike coefficients **size** zernike coefficients array size.
Returns:	**giapi::status::OK** if the zernikes were offloaded correctly to the PCS or **giapi::status::ERROR** if there was a problem in the operation.
Throws:	**GiapiException** in case there is an error accessing the GMP to post the PCS update

6.5 Access to the TCS Context

Production of WCS information is done in the instrument software by using values in a structure fetched from the TCS called the TCS Context. The WCS conversion and TCS Context is described in [5]. An instrument-specific calculation is then done to produce the values for the WCS header items. These calculations usually involve coordinate transformations that we support though C-library function calls.

The GIAPI C++ Language glue provides the following method in the `GeminiUtil` class that can be called to fetch the TCS context:

getTcsContext

Signature:	`static int getTcsContext(TcsContext& ctx, long timeout) throw (GiapiException)`
Description:	Provides the TCS Context information at the time of the call. The TCS context provides information about the TCS that are needed to perform WCS conversions.
Parameters:	**ctx** Reference to the `TcsContext` structure. The content of this structure will be filled in by this call. **timeout** Time in milliseconds to wait for the TCS context to be retrieved. If not specified, the call will block until the GMP replies back or an exception occurs.
Returns:	**giapi::status::OK** if the TcsContext was filled in properly. **giapi::status::ERROR** if there was an error getting the TcsContext.
Throws:	**GiapiException** in case there is an error accessing the GMP to obtain the TCS context

`getTcsContext` can also be used when the instrument needs to make occasional calculations unrelated to WCS that require much of the information available in the TCS context.

Table 12 contains the elements present in the `TcsContext` structure in the GIAPI C++ Language glue. This structure is defined in <giapi/giapi.h> header file. All the elements are doubles. It is possible to provide a subset of the most useful information depending on builder need.

Context Index	Context Item Name	Description
0	time	Time stamp (Gemin Raw Time)
1-3	x, y, z	Cartesian elements of mount pre-flexure az/el
4	tel.fl	Telescope focal length (mm)
5	tel.rma	Rotator mechanical angle (rads)
6 7	tel.an tel.aw	Azimuth axis tilt NS (rads) Azimuth axis tilt EW (rads)
8	tel.pnpae	Az/el non-perp (rads)
9 10	tel.ca tel.ce	Left-right collimation (rads) Up-down collimation (rads)

11-24	aoprms	Target independent apparent-to-observed parameters
25-30	m2xy	M2 tip/tilts
31 32	po.mx po.my	Mount pointing origin in X Mount pointing origin in Y
33 34 35 36 37 38	ax ay bx by cx cy	Source chop A pointing origin in X Source chop A pointing origin in Y Source chop B pointing origin in X Source chop B pointing origin in Y Source chop C pointing origin in X Source chop C pointing origin in Y

Table 12: Contents of the TCS Context as received directly from the TCS

7 Using Gemini Services

Gemini Services are features of the Gemini environment that most instruments need. This section describes the currently available Gemini services. This discussion parallels the discussion in section 12 of [2].

7.1 General Overview

The following is a list of the current supported services in the GIAPI, and a reference to the sections in which they are described:

1. Logging Service (Section 7.2). Methods to support process logging and system logging
2. Access to the observatory time (Section 7.3). Allows access to the current time that is standardized and synchronized across the observatory
3. Access to GIAPI properties (Section 7.4). Provides access to configuration attributes to the builder through the GIAPI properties interface.

The methods to access the GIAPI Gemini Services are provided in the `ServicesUtil` class. This class is defined in the `<giapi/ServicesUtil.h>` header file.

7.2 Logging Service

There are two supported logging approaches in the GIAPI: process logging and system logging. Process logging is the type of logging used to debug a process. System logging is a service supported by the GMP that allows any process to contribute to a system log.

7.2.1 Process Logging

GIAPI standardizes the usage of the Apache log4xx library for logging that is local to a single process. log4cxx is C++ port of the Java Log4j library. log4cxx attempts to mimic log4j usage as much as the language will allow and to be compatible with log4j configuration and output formats. Information about log4xx can be found in [7].

7.2.2 System Logging

GIAPI allows any process in the instrument to send a logging message that will then be merged into an instrument-wide log maintained by the GMP. As with Process Logging, System Logging will be tied to log levels and will be configurable at runtime to enable or disable logging specific log levels.

In the C++ Language Glue, this capability is provided by the `systemLog` call in the `ServicesUtil` class.

systemLog

Signature:	`static void systemLog(log::Level level, const std::string & msg)` `throw (GiapiException)`
Description:	Logs a message that will be merged into an instrument-wide log, using the given log level.
Parameters:	**level** The logging level identifier for this log message, such as `giapi::log::WARNING`.
Returns:	**void**

Throws:	**GiapiException** in case there is an error accessing the GMP to log the message

The supported logging levels are `INFO`, `WARNING` and `SEVERE`. These are enumerated types defined in the `log` namespace. They are declared in the `<giapi/giapi.h>` header file.

For example the following call:

```
ServicesUtil::systemLog(log::INFO, "WCS information was incomplete.  Missing CTYPE1");
```

will log the message at the INFO level. If the instrument-wide log is configured to log at the INFO level, this message will then be merged into the log.

7.3 Access to Observatory Time

The C++ Language glue provides one method to access the time regardless of the internal implementation (either software or hardware based). The single required method returns the UTC time as the number of milliseconds since January 1, 1970. The following is the method description:

getObservatoryTime

Signature:	`static long64 getObservatoryTime() throw (GiapiException)`
Description:	Returns the current observatory time in milliseconds. The granularity of the returned value depends on the underlying operating system and may be larger.
Parameters:	**none**
Returns:	Number of milliseconds between the current observatory time and midnight, January 1, 1970 UTC as a 64-bit long integer
Throws:	**GiapiException** in case there is an error accessing the GMP to get the observatory time

Since a 64-bit integer is needed to represent the returned value by this call, a `long64` type is used. Since 64-bit signed integers are not part of the ANSI C++ standard, this definition is compiler specific. The definition is in the `<giapi/giapi.h>` header file.

7.4 GIAPI Configuration Properties

The GIAPI can provide configuration attributes to the builder through the GIAPI properties interface. A property is a string value associated with a string key. In the C++ Language glue, the following is the method provided in the `ServicesUtil` class to access GIAPI properties.

getProperty

Signature:	`static std::string getProperty(const std::string &key,`
	`long timeout = 0)`
	`throw (GiapiException)`

Description:	Returns the GIAPI property indicated by the specified key.
Parameters:	**key** the name of the GIAPI property to request **timeout** time (in milliseconds) to wait for the GMP to reply back a property. If not specified, it will block until the GMP replies.
Returns:	The string value of the GIAPI property, or an empty string if there is no property with that key.
Throws:	**GiapiException** in case there is an error accessing the GMP to get the required property. A **TimeoutException** is generated in case the timeout specified expires without having an answer from the GMP

An example:

```
std::string host = ServicesUtil::getProperty("GMP_HOST_NAME");
```

In this example, the `getProperty` call returns the host name of the machine running the GMP.

The following are the keys currently defined. These will be expanded in the future as needed.

Property Key	Description
GMP_HOST_NAME	Host name of the machine where the GMP is running
DHS_ANCILLARY_DATA_PATH	The path to use to write ancillary data files.
DHS_SCIENCE_DATA_PATH	The path to use to write science data files.
DHS_INTERMEDIATE_DATA_PATH	The path to use to write intermediate data files.
DEFAULT	The default key, associated to the default value. The default value is an empty string.

Table 13: GIAPI Properties in the C++ Language Glue

8 Providing Science Data to Gemini

This section discussed the C++ GIAPI methods related to writing instrument science data. This section provides the C++ information for the features described in section 13 of [2].

8.1 General overview

The transfer of data from instruments to the DHS has been simplified for Aspen instruments by using a high-performance Network Attached Storage (NAS) system at both telescopes known as the Gemini Data Storage Network (GDSN). Instruments directly write data to a shared disk, reducing the complexity of the data transfer code builders must produce.

The actual writing of a dataset or ancillary file is simple: use the FITS library to write internal format data as a valid FITS file (see [3]) on the GDSN. In C++ the library chosen for this purpose is the CFITSIO or CCFITS [4](http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html). Files will need to be specified and written to use the capabilities of this standard library.

The shared disk mount point where data files must be written is a GIAPI property. Table 13 lists all the properties that can be used to retrieve different mount points. For instance, the following call:

```
std::string path = ServicesUtil::getProperty("DHS_SCIENCE_DATA_PATH");
```

will retrieve the path to store science data files. The file name is generated by the **observe** sequence command `DATA_LABEL` parameter (see section 5.3.1); the instrument normally just adds the .fits extension to its value.

The other step required to integrate dataset writing with Gemini is to notify that the execution of an **observe** sequence command has reached and started specific phases of the process.

The C++ Language glue provides a `postObservationEvent` method to send notifications (called *Observation Events*) when specific parts of the execution of the **observe** sequence command are reached. Section 8.2 discusses how to send observation events through the C++ Language glue.

In addition to normal science datasets, GIAPI provides methods to support handling of ancillary files and intermediate files. These are covered in section 8.3 and 8.4 respectively.

The following table summarizes the classes and data types used in the GIAPI C++ language glue to support data handling.

Class/Type	Description	C++ header
`DataUtil`	Provides methods to send observation events and notify of ancillary and intermediate files	<giapi/DataUtil.h>
`data::ObservationEvent`	Enumerated type defining the different *Observation Events* supported by the C++ Language glue.	<giapi/giapi.h>

Table 14: Classes and data types to support data handling

8.2 Sending Observations Events with GIAPI

The observation events supported in the C++ Language glue are listed in Table 15. The table has a complete description of each phase and indicates some of the actions that Gemini takes in response to the delivery of the events. These events are represented as enumerated types defined in the `data` namespace in the `<giapi/giapi.h>` header. Therefore, a fully qualified name of one event would be `giapi::data::OBS_START_ACQ`.

Event Time	GIAPI OEvt	Description
T1	OBS_PREP	Observation Event sent as instrument starts preparation for starting acquisition of a dataset.
T2	OBS_START_ACQ	Observation Event sent just before data acquisition starts.
T3	OBS_END_ACQ	Observation Event send when the requested acquisition has completed.
T4	OBS_START_READOUT	Observation Event indicates that the data is being transferred from the detector or other activities needed to write data.
T5	OBS_END_READOUT	Observation Event indicates readout or write preparations have completed.
T6	OBS_START_DSET_WRITE	Observation Event indicates that the instrument has started writing the dataset to the GDSN.
T7	OBS_END_DSET_WRITE	Observation Event indicates that the instrument has completed writing the dataset to GDSN.

Table 15: Observation events in the C++ Language Glue

The observation events in Table 15 are listed in the order they are sent during the execution of an **observe** sequence command. A detailed description of observing events can be found in [2].

As each phase is reached, the instrument sends an Observation Event to Gemini with the `postObservationEvent` method available in the `DataUtil` class.

postObservationEvent

Signature:	`static int postObservationEvent(data::ObservationEvent event, const std::string & datalabel) throw (GiapiException)`
Description:	Send an observation event associated to the given data label to the GMP. This method should be called by the instrument at each phase of the execution of an **observe** sequence command.
Parameters:	**event** The observation event to be sent, such as `data::OBS_START_ACQ` **datalabel** the data-label that is being generated by the observe command.
Returns:	**giapi::status::OK** if the event was dispatched properly or **giapi::status::ERROR** if there is a problem.
Throws:	**GiapiException** in case there is an error accessing the GMP to post the given observation event

As an example:

```
DataUtil::postObservationEvent(data::OBS_START_ACQ, datalabel);
```

Where `datalabel` is the label obtained in the `SequenceCommandHandler` of the **observe** sequence command as described in section 5.3.

8.3 Ancillary Files and GIAPI

Ancillary files are stored as FITS files in the GSDN. In the C++ Language glue, the instrument will write the ancillary file to the GDSN and post an Observation Event notifying Gemini of the new file. The event can be posted at any time, not just during an **Observe**.

This is the method in the C++ Language glue to post ancillary files events:

postAncillaryFileEvent

Signature:	`static int postAncillaryFileEvent(const std::string &filename, const std::string &datalabel) throw (GiapiException)`
Description:	Send an ancillary file event associated to the given datalabel to the GMP.
Parameters:	**filename** name of the ancillary file **datalabel** the data-label generated by an **observe** sequence command that will be associated to this ancillary file
Returns:	**giapi::status::OK** if the event was dispatched properly or **giapi::status::ERROR** if there is a problem.
Throws:	**GiapiException** in case there is an error accessing the GMP to post the given ancillary file event

Here is an example call:

```
DataUtil::postAncillaryFileEvent("AOStats-20100222.fits", "GN-2010B-Q-2-1-003");
```

8.4 Support for Intermediate Files

During the execution of an **observe** sequence command, the GIAPI C++ Language glue allows the builder to notify Gemini of the creation of intermediate FITS files. The method for alerting Gemini works similarly to the ancillary file method in the previous section.

postIntermediateFileEvent

Signature:	`static int postAncillaryFileEvent(const std::string &filename, const std::string &datalabel, const std::string &hint) throw (GiapiException)`
Description:	Send an intermediate file event associated to the given data label to the GMP and with the given hint.
Parameters:	**filename** name of the intermediate file **datalabel** the data-label generated by an **observe** sequence command that will be associated with this intermediate file **hint** The hint can be used to indicate different types of intermediate files and is instrument dependant.
Returns:	**giapi::status::OK** if the event was dispatched properly or **giapi::status::ERROR** if there is a problem.
Throws:	**GiapiException** in case there is an error accessing the GMP to post the given ancillary event



The hint can be used to indicate different types of intermediate files. The hints are instrument dependent and should be published in ICD 16. For instance, a builder may wish to publish an "engineering" file for viewing in engineering mode and a display tool could be configured to only display "engineering" intermediate files. Here is an example:

```
DataUtil::postIntermediateFileEvent("NIFS-Coadds-20100222.fits",
                                    "GN-2011B-Q-3-1-043", "coadds");
```

The hint idea is preliminary and may require changes during initial use.

9 Error Handling and Fault Tolerance in the GIAPI C++ Language Glue

The GIAPI C++ Language glue provides some mechanism to facilitate handling of error conditions. The usage of these methods is intended to provide fault tolerance and auto-recovery options, in particular when the communication to the GMP is broken.

9.1 General Overview

Utility methods are provided in the GIAPI to register user provided error handlers that will be invoked when a communication problem is found. The methods to access the GIAPI Utility services are provided in the `GiapiUtil` class. This class is defined in the `<giapi/GiapiUtil.h>` header file.

The following are the main classes the instrument code will need to work with to access these utility methods and the C++ headers where they are defined.

Class/Types	Description	C++ header
`GiapiUtil`	Collection of static methods to help dealing with errors and provide fault tolerance options.	<giapi/GiapiUtil.h>
`GiapiErrorHandler`	Interface for GIAPI Error handlers	<giapi/GiapiErrorHandler.h>
`giapi_error_handler`	Function pointer to represent an error handler	<giapi/giapi.h>

Table 16: Main classes to provide interaction with the utility methods available in the C++ Language glue

9.2 Error Handlers for GMP communication problems

The `GiapiUtil` class provides a mechanism for client code to register handlers that will be invoked whenever the GMP communication is lost. This happens for instance if for some reason the GMP crashes and needs to be restarted. By using these methods, it is possible for a running application to recover graciously in case of this event.

If the communication to the GMP is lost for some reason, the GIAPI automatically will get an event and will try to re-establish the communication. Once the communication is available again, the application needs to re-establish the previous subscriptions it depended upon. For instance, if an application registered to receive the INIT sequence command, via the `CommandUtil::subscribeSequenceCommand()` method, that call needs to be invoked again if the GMP crashes and needs to be restarted.

To facilitate this, the `GiapiUtil` class provides two mechanisms. The first one is an object-oriented approach, whereas the second is a more procedural approach.

9.2.1 Object-oriented approach to deal with GMP communication problems

The first mechanism uses an interface, called `GiapiErrorHandler`. This interface allows client code to implement a method that will be invoked every time the GMP communication needs to be reset. This interface has only one method that needs to be implemented, called `onError`. The following is the signature of the `onError` method in this interface.

onError

Signature:	`void onError(void)`
Description:	Callback invoked when there is a problem with the GIAPI Language Glue trying to communicate with the GMP. Implementations of this class should put into this call all the activities that need to be performed in case there is a problem with the GMP, especially if the GMP needs to be restarted.
Parameters:	**void**
Returns:	**void**

The `pGiapiErrorHandler` is a smart pointer definition to a `GiapiErrorHandler` implementation.

In the `GiapiUtil` class, the `registerGmpErrorHandler` method allows to register a `GiapiErrorHandler` with the GIAPI. This method will be invoked whenever the GMP connection needs to be reestablished by the GIAPI. The following is the signature of this method:

registerGmpErrorHandler

Signature:	`static void registerGmpErrorHandler (pGiapiErrorHandler handler) throw (CommunicationException)`
Description:	Register and error handler object (as a smart pointer) that will be invoked when a GMP connection problem happens and the connection is recovered. In the event of a communication problem with the GMP, the GIAPI library will attempt to restore the connection to the GMP. Once the connection is available again, all the handlers registered through this method will be invoked. The same handler can't be registered more than once.
Parameters:	**handler** The handler that will be invoked when the communication to the GMP is lost and later recovered.
Returns:	**void**
Throws:	**CommunicationException** in case there is an error accessing the GMP when the call is made

A normal usage pattern of this method is for a client to create an implementation of the `GiapiErrorHandler`, and make sure the `onError()` method calls all the methods that makes subscriptions with the GMP (like the `subscribeApply`, `subscribeSequenceCommand`, `subscribeEpicsStatus`, etc).

9.2.2 A procedural approach to deal with GMP communication problems

In case is not convenient to use a C++ interface, it is also possible to register an error handler through a function pointer. In the <giapi/giapi.h> header the following definition exists:

```
typedef void (*giapi_error_handler)(void);
```

Client code can register a simple C-style function based on that definition that will be invoked whenever a communication problem with the GMP occurs. The usage pattern is similar to the object-oriented approach. A similar method in the `GiapiUtil` class is provided to register these function pointers with the GIAPI:

registerGmpErrorHandler

Signature:	`static void registerGmpErrorHandler (giapi_error_handler handler) throw (CommunicationException)`
Description:	Register and error handler (as a function pointer) that will be invoked when a GMP connection problem happens and the connection is recovered. In the event of a communication problem with the GMP, the GIAPI library will attempt to restore the connection to the GMP. Once the connection is available again, all the handlers registered through this method will be invoked. The same handler can't be registered more than once.
Parameters:	**handler** A function pointer that will be invoked when the communication to the GMP is lost and later recovered.
Returns:	**void**
Throws:	**CommunicationException** in case there is an error accessing the GMP when the call is made

10 Exceptions in the GIAPI C++ Language Glue

As described in the previous sections, several methods in the GIAPI C++ Language glue throw exception when an operation cannot complete. This section describes the different exceptions in use in the GIAPI.

10.1 Exception Classes in the GIAPI C++ Language Glue

Exceptions in the GIAPI C++ Language glue are modeled using the standard C++ `std::exception` class as the base class. The following diagram depicts the class hierarchy used in the GIAPI to model the exceptions.

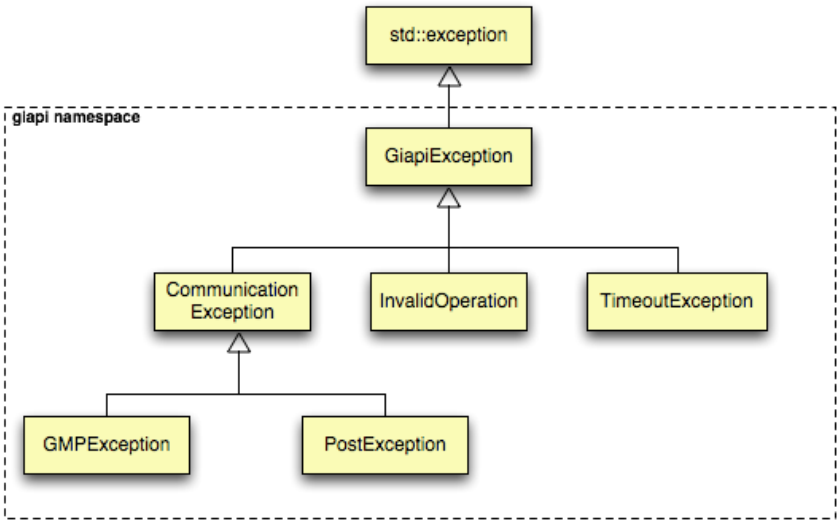


Figure 4: GIAPI Exception Hierarchy

The `GiapiException` is the base class for all the exceptions in the GIAPI and the one that the high-level methods in the library will throw. Client code could only handle this exception if no more information is needed. If more information is needed, client code can handle the specific exceptions in the exception hierarchy to figure out the exact cause of a problem.

The definitions of all the exceptions in the GIAPI C++ Language glue are in the header `<giapi/giapiexcept.h>`. All the exceptions are in the `giapi` namespace. The `GiapiException` base class defines mechanisms to provide a textual description of the cause of a problem. The following is a summary of these methods:

setMessage

Signature:	`void setMessage(const std::string & msg, ...)`
Description:	Set the cause for this exception
Parameters:	**msg** A textual description of the cause of the exception **variable** Parameters to format the message into a string

setMessage

Signature:	`void setMessage(const std::string & msg)`
Description:	Set the cause for this exception
Parameters:	**msg** A textual description of the cause of the exception

getMessage

Signature:	`std::string getMessage() const`
Description:	Gets the text message for this exception
Parameters:	**None**

In addition, the constructor for the exception can take a string as an argument to describe the cause of the exception.

The following table contains the description of all the other exceptions and when they are thrown.

Exception	Description
`CommunicationException`	These exceptions are generated when some communication problem happens between the GIAPI library and the Gemini Master Process.
`InvalidOperation`	Thrown if the requested operation is not valid in the current context.
`GMPException`	A communication exception caused by a problem trying to contact the Gemini Master Process
`PostException`	A communication exception raised when some condition prevents a post operation to complete
`TimeoutException`	A timeout expired when waiting for an operation to complete, usually involving a communication with the Gemini Master Process

Table 17: Exceptions in the GIAPI library