GIAPI Design and Use

Version 08

Kim Gillies, Arturo Núñez

GIAPIUse-08292006/08
Created: September 10, 2007
Modified: November 18, 2014



Change Record

Version	Date	Author	Notes
1	October, 30 2006	Kim Gillies	• Draft Version – Released 11/30/2006 – KG.
2	23 January 2008	Kim Gillies	• Second Release – Updated for GPI group comments.
3	02 February 2009	Arturo Núñez	• Updated to make reference to the GIAPI Builder Requirements document and fixed some minor typos
4	03 March 2011	Arturo Núñez	• Removed indication that the Reboot sequence command will park the instrument first
5	07 November 2011	Arturo Núñez	• Updates for new instruments – Draft version
6	03 February 2012	Nicolás A. Barriga	• Added engineering commands
7	17 November 2014	Arturo Núñez	• Replaced RedHat with CentOS. Added performance requirement for sequence command (REQ-GIAPI-SC-19)
8	18 November 2014	Arturo Núñez	• Updated REQ-GIAPI-SC-19 based on Systems Engineering review.

Table of Contents

1 Introduction 1

1.1 Document Purpose 1

1.2 GIAPI Document Package 1

1.3 Intended Readership 2

1.4 Thoughts on Building an Instrument for Gemini 2

1.5 Disclaimer for Software Interfaces 3

1.6 Conventions 3

1.7 Acronyms 3

1.8 Reference Materials 4

2 GIAPI Philosophy 5

2.1 Guidelines for Changes 5

2.1.1 Simplify the Gemini Environment 5

2.1.2 Allow Builders More Freedom 5

2.1.3 Simplify the Instrument Integration Effort 5

2.1.4 Clarify Software Responsibilities 5

2.1.5 Reduce Software Costs in Effort and Equipment 6

2.1.6 Increase Quality and Rigor 6

2.1.7 Provide Timely and Appropriate Levels of Support 6

2.2 Splitting Builder and Gemini Responsibilities 6

2.2.1 Gemini Role 6

2.2.2 Builder Role and Responsibilities 6

3 Gemini Instrument Background 6

3.1 Instrument Sequencer 7

3.2 Components Controller 8

3.3 Detector Controller 8

3.3.1 Internal Data Processor 9

4 GIAPI Functional Overview 9

4.1 Commands, Status and Science Data 9

4.2 Instrument Integration with the Gemini Telescope Software and Hardware 10

5 Computer Hardware Choices 11

5.1 High-Level Physical Layout of a Gemini Instrument 11

5.2 Hardware Choices 12

5.2.1 NEBS Compliance 13

5.2.2 Two Phase Hardware Planning 13

5.2.3 Disk Drive Considerations 13

5.2.4 Computer Bus Considerations 14

5.3 Network Environment 14

5.4 Network Storage System 14

6 Operating System Choices 14

7 GIAPI Architectural Overview 14

7.1 Implementation Overview 16
7.1.1 Java Message Service 16
7.1.2 Open Systems Gateway Initiative Framework 17
7.1.3 GIAPI Implementation Using JMS and OSGi Framework 17
7.1.4 Performance Integration Issues and Risks 19

7.2 Language Glue Choices 19
7.2.1 The Need for Standardization 19

7.3 GIAPI Infrastructure Enhancements 19
7.3.1 Use of Enhancements 19

8 Library Usage *19*

8.1 Naming of items in the GIAPI 20
8.1.1 Component Name Options 20

9 Providing Status to Gemini *21*

9.1 Uses for Status 21
9.1.1 Status System Performance 21

9.2 Status Items 21
9.2.1 What Values Should be Published? 22

9.3 Status Item Alarm Conditions 22

9.4 Status Items and Instrument Health 23

9.5 How To Use Status in the GIAPI 24
9.5.1 Builder Responsibilities 24
9.5.2 Status Item Naming 24
9.5.3 Public and Private 25
9.5.4 Private Status and Private Status Subscribing E 25
9.5.5 Timely Status Updates 26
9.5.6 Scenarios 26

9.6 GIAPI Functionality Summary 27
9.6.1 GIAPI Infrastructure Enhancement Functionality Summary 27
9.6.2 Usage Patterns 27

10 Receiving Sequence Commands from Gemini *27*

10.1 Command Definitions 27

10.2 Action Command Model 28

10.3 Sequence Commands 28
10.3.1 Instrument Initialization, Setup and Shutdown 28
10.3.2 Handling Reboot on Linux 29
10.3.3 Observation Preparation Sequence Commands 31
10.3.4 Apply Sequence Command 32
10.3.5 Data Acquisition Sequence Commands 33
10.3.6 Engineering Commands 34
10.3.7 General Notes on Sequence Commands 34

10.4 Sequence Command Concurrency 34

10.5 Handling Gemini Sequence Commands in GIAPI 35

10.6 Sequence Command Handlers 35

10.7 Subscribing to Sequence Commands other than Apply 37

10.8 Subscribing to the Apply Sequence Command 37
10.8.1 Configurations and Parameter Naming 37

10.8.2 Subscribing to Apply in GIAPI 38
10.9 Updating Gemini with Action Status 38
10.10 Dealing With Apply Errors 39
10.10.1 Acceptance Errors 39
10.10.2 Immediate Completion 39
10.10.3 Ongoing Actions 39
10.11 Suggested Usage Patterns 40
10.11.1 Single Preset Handler 40
10.12 Private Commands E 41
11 Interacting with Gemini Systems 41
11.1 Receiving Status Information From a Gemini System 41
11.1.1 GIAPI EPICS Status Monitoring Functionality 42
11.1.2 Monitoring Scenario 43
11.2 Instrument OIWFS Updates from AGCS 43
11.3 Making Information Available to the TCS 43
11.3.1 Offloading To the Primary Mirror 43
11.4 World Coordinate System Interactions 43
11.5 Controlling the Gemini Secondary 44
12 Using Gemini Services 45
12.1 Logging Service 45
12.1.1 Process Logging 45
12.1.2 System Logging 45
12.2 Access to Observatory Time 45
12.2.1 Software-Only System 46
12.2.2 Hardware-Based System 46
12.2.3 GIAPI Time Support 46
12.3 GIAPI Properties 46
13 Providing Science Data to Gemini 46
13.1 The Gemini Data Environment 47
13.2 Writing Datasets 48
13.2.1 Datasets as FITS Standard Files 48
13.2.2 Writing FITS Headers At Gemini 49
13.2.3 Writing FITS Headers During Engineering/Testing 49
13.2.4 GDSN Issues 49
13.2.5 Storage for Instruments 50
13.3 Science Data HDU Definition Process 50
13.4 GIAPI Support for Writing Datasets and Ancillary Files 50
13.4.1 Observation Events 50
13.4.2 Phases and Observe Observation Events in GIAPI 50
13.5 Ancillary Files and GIAPI 52
13.6 Intermediate Files for Display 52
14 Development Process 53
14.1 Iterative Development 54

15 Testing with GIAPI 55
16 Gemini Concurrent Instrument Development 55
16.1 Acceptance Test and Engineering User Interface 56
16.2 Operations Control User Interface 56
16.3 Observatory Control System Integration 56
17 Tools and Releated Programming Details 56

1 Introduction

This document provides a description of the design and use of 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 document describing the GIAPI for Gemini instruments is *Guidelines for Designing Gemini Instrument Software* [1] (referred to as the Instrument Guidelines or IG).

This document is the top-level document in the GIAPI documentation package (see Section 1.2). The goal is for this document to be the README FIRST for GIAPI. The design and use of the GIAPI is presented in considerable detail. Requirements and patterns for its use are described to clarify how responsibilities are balanced between Gemini and the instrument builder.

The software issues related to data processing and integrating with the Gemini data processing pipeline are covered in another document and companion ICD [22] and [23].

1.1 Document Purpose

The purpose of this document is to define the architecture of the GIAPI, to help builders understand how the GIAPI is used to integrate instruments with the Gemini software system, and to specify the kinds of things Gemini requires of instrument software. This document does the following:

- Provides details of the GIAPI in enough depth to allow instrument builders to design their software through the PDR development phase.
- Provides guidance in structuring instrument software in more detail than in IG.
- Provide a list of the things that must be done to integrate with Gemini.
- Provide a "how to" for builders with sections on each of the things that must be done.
- Provide information on the non-technical issues related to instrument software development for Gemini including use of tools, operating systems, hardware choices and configuration control.
- Provide details on the collaborative development model that is to be used with Gemini Instrument software.

Examples of method calls are shown and discussed throughout this document in a way that is object-oriented and easily visualized in a target language (currently Java and C++). While the document describes procedures in a way that is somewhat independent of any specific programming language, each language API provides a way to do each of the procedures.

1.2 GIAPI Document Package

This document is one document in the GIAPI Document Package that is shown in Figure 1. This document is shown surrounded by a box. The other primary reference documents that flow directly from this document are the language glue ICDs and the GIAPI Builder Requirements.

The plan is to provide GIAPI support in two languages: C++ (currently available) and Java. There will be an ICD that describes the specifics of the API for each supported language [19] and [20].

The GIAPI Builder Requirements Document summarizes the things that the builder must do and steps for testing requirements compliance.

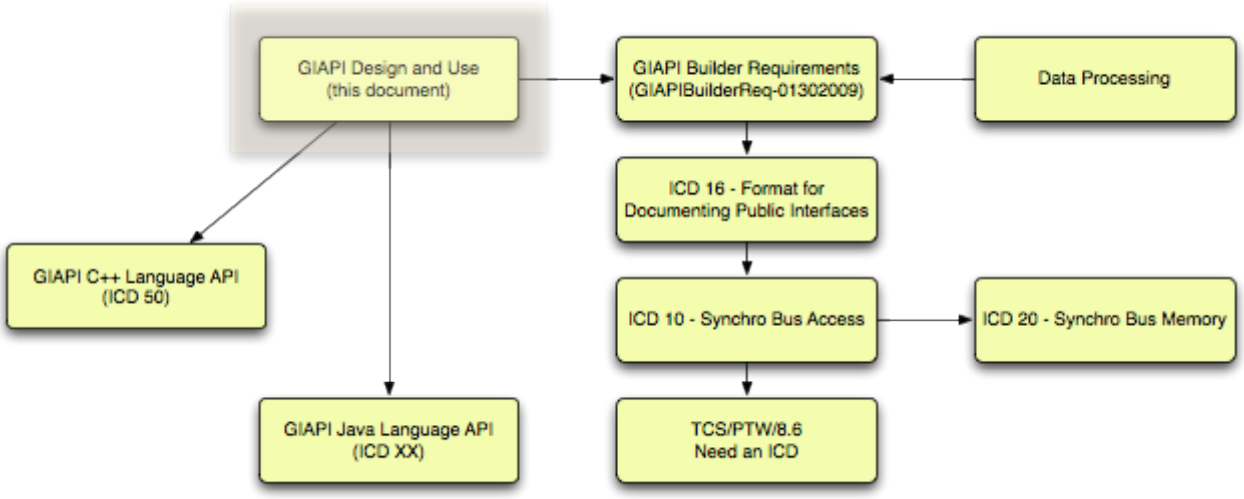


Figure 1: GIAPI Documentation Tree

Below the GIAPI Requirements document are additional ICDs that provide information on specific topics. ICD 16, Format for Documenting Public Interfaces, describes what should be included in the public interface for the instrument, and how the public interface should be defined. ICD 10 has been updated to indicate how the Synchro Bus should be used from x86/Linux-based computer hardware. TCS/PTW/8.6 is a document describing the support of World Coordinate Systems.

Throughout this document, references to the GIAPI Builder Requirements are made using the id of each requirement. The ids have the form GIAPI-REQ-SYS-XX, where SYS is replaced with a short code that represents the aspect of the system the requirement affects. Details are found in [21].

1.3 Intended Readership

The intended audience for this document is *groups who are writing software or design review documents for Gemini instruments*. The document is targeted specifically towards programmers or other specialists who understand software and who must design or estimate the size and cost of the software component of a Gemini instrument.

This document attempts to provide enough information about the Gemini software system such that any senior software specialist can understand what must be done for a Gemini instrument. The document assumes the reader has a solid background in programming and astronomy instrument development. This document uses the concepts and nomenclature that are part of the IG document and assumes the reader's familiarity with that document since it provides the background about Gemini that is assumed in this document.

1.4 Thoughts on Building an Instrument for Gemini

Building a science instrument for Gemini is a huge undertaking costing millions of dollars and years of people's work lives. The instrument software effort, no matter how large or small, has a similar and important role to play in the project—it's software that integrates all the pieces to provide a system that accomplishes the science goals. The other engineers don't like to admit it, but it's software that makes or breaks an instrument and brings it to life.

This document has a lot of information, and it takes quite a bit of time to understand all the facets of Gemini. Some things about Gemini will no doubt seem sub-optimal or even silly. Many software reviews have suggested the following points should be in the mind of the Gemini instrument developer.

- Gemini is the product owner and providing software that does what is needed is an engineering task.
- The instrument software isn't a research project or a place to use untested ideas. Gemini isn't something that is added on at the end of the project after all the *good* software is done.
- Do what the project wants you to do. Use the terms and concepts of Gemini — don't invent or use other concepts because you can. Explain things as simply as possible. Do what the project wants you to do.

- The only purpose for building the instrument software is to enable a Gemini instrument to work properly in the Gemini environment.

1.5 Disclaimer for Software Interfaces

This version of this document is based on the initial version of the C++ GIAPI used for the Gemini Planet Imager, GPI. The design and use of GIAPI presented here is as accurate as it can be at this time in its development. We believe the API to be stable and usable at this time. But the design, overall approach, and procedures are correct and should change very little (although they may also change a bit.) Our goal is to not change for the sake of change but to be flexible to changes that help provide a better product.

Our intention in this document is to allow current projects to move forward with their design work by detailing how the GIAPI works and how it is used in the builder code.

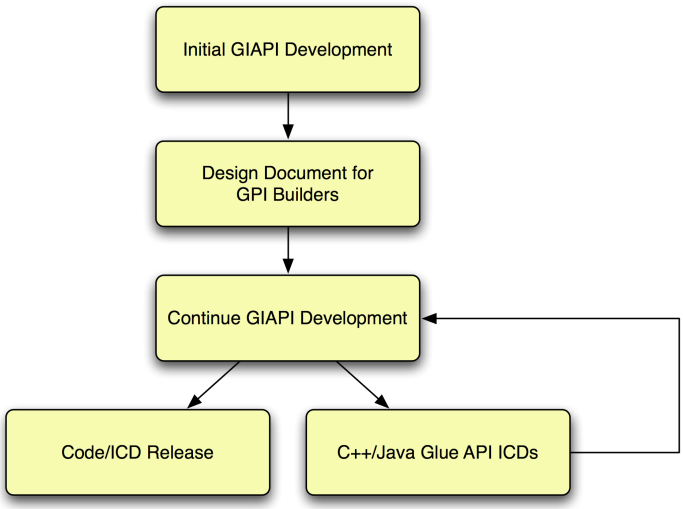


Figure 2: Ordering of development for GIAPI-based instrument software products

The initial plan was to complete the first version of the GIAPI software and then provide documentation based on the results. The idea being that putting code in the hands of developers is of more value than paper that talks about it. Our experience with Gemini instruments and our control system allows this code-quickly approach. We are also promoting a more agile approach to development that focuses on delivery of working code features over documentation artifacts.

Figure 2 shows the current plan. The initial version of the GIAPI has been completed and important concepts have been shown to work (top of figure). Documentation has been produced that benefits early instrument builders. This is shown in the second oval. This has been a useful task because it has required us to consider several issues related to EPICS integration and telescope services. Following this step code development will continue along with the development of the other top-level documents including the specific language ICDs.

1.6 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.

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

1.7 Acronyms

ACM	Action Command Model
AGCS	Acquisition and Guide Control System
ATEUI	Acceptance Test and Engineering User Interface
AO	Adaptive Optics
API	Application Programmer Interface
ASP	Aspen Software Process
CC	Components Controller

CDR	Critical Design Review
DC	Detector Controller
DHS	Data Handling System
EPICS	Experimental Physics and Industrial Control System
GDSN	Gemini Data Storage Network
GIAPI	Gemini Instrument Application Programmer Interface
HDU	Header Data Unit (FITS standard nomenclature)
HL	High Level
HLSDG	High Level Software Development Group
ICD	Interface Control Document
ICS	Instrument Control System
IG	Guidelines for Designing Gemini Instrument Software
IOC	Input-Output Crate
IRIG-B	Inter-Range Instrumentation Group – Time Format B
IS	Instrument Sequencer
JAR	Java ARchive file
JVM	Java Virtual Machine
GMP	Gemini Master Process
NAS	Network Attached Storage
NEBS	Network Equipment Building System
OIWFS	On-Instrument Wavefront Sensor
OCS	Observatory Control System
OEVT	Observation Events
ODB	Observing Database
OLDP	On Line Data Processing
OS	Operating System
OT	Observing Tool
PCS	Primary Control System
PDR	Preliminary Design Review
PI	Principal Investigator
SCH	Sequence Command Handler
SCS	Secondary Control System
SPMP	Software Project Management Plan
SSA	System Support Associates
TBD	To Be Determined
TCS	Telescope Control System
TLC	Top Level Computer
WCS	World Coordinate System

1.8 Reference Materials

[1] Guidelines for Designing Gemini Instrument Software, Kim Gillies, Arturo Núñez, InstSoftGuidelines-03072004-7

[2] Java Message Service home site: http://java.sun.com/products/jms/index.jsp.

[3] Java Message Service standard specification: http://java.sun.com/products/jms/docs.html.

[4] ActiveMQ Home Site: http://www.activemq.org.

[5] OSGi Alliance Home Site: http://www.osgi.org.

[6] Equinox OSGi R4 implementaiton: http://www.eclipse.org/equinox.

[7] Oscar OSGi R3 implementation: http://oscar.objectweb.org.

[8] Apache Felix OSGi R4 implementation: http://cwiki.apache.org/FELIX/index.html.

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

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

[11] ICD 7b, ICD Subsystem Interface

[12] Gemini ICD 9, *EPICS Time Bus Driver*, Version ICD 09/01, September 5, 1996.

[13] RFC 1305, NTP V3 Specification, http://www.ietf.org/rfc/rfc1305.txt?number=1305.

[14] Agile Modeling and Documentation, http://www.agilemodeling.com
[15] Scrum Information: http://www.controlchaos.com/about/ and this page and external links: http://en.wikipedia.org/wiki/Scrum_(development)
[16] Scrum in Five Minutes: http://www.softhouse.se/Uploades/Scrum_eng_webb.pdf.
[17] *Agile Project Management with Scrum*, Ken Schwaber, Microsoft Press, 2004.
[18] FITSIO Home Page, http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html
[19] ICD 50, GIAPI C++ Language Glue, Arturo Núñez, Kim Gillies, GIAPIC++ICD50-02042008/04
[20] GIAPI Java Language ICD, TBW
[21] GIAPI Builder Requirements Document, Arturo Núnez, GIAPIBuilderReq-01302009/01
[22] Guidelines for Instrument Data Processing, TBW
[23] GIAPI Data Processing ICD, TBW

2 GIAPI Philosophy

Gemini has spent a tremendous amount of time and money developing the telescope systems and instruments that are currently in service and under development. While these systems are working well, many lessons were learned from this effort, and it is critical that the lessons learned be captured and applied to future instrument development. Computing equipment and operating systems have drastically improved over the last few years providing options that were impossible a few years ago.

The GIAPI is the result of providing a broad set of options in the IG document, observing the choices builders made during feasibility and conceptual designs, and limiting our support to what was an unconscious consensus by builders.

This section recaps the motivation for the changes being introduced for Gemini instrument development. The guidelines are primarily pragmatic rather than technical.

2.1 Guidelines for Changes

The following sections describe the guidelines for the changes.

2.1.1 Simplify the Gemini Environment

Builders need to understand the Gemini environment. In many ways, it is unlike other observatories with the emphasis on observation planning and queue observing. Instrument performance requirements are demanding and Gemini exposes the builder to new issues such as embedded wavefront sensors and the World Coordinate System. GIAPI tries to minimize the impact of these requirements.

2.1.2 Allow Builders More Freedom

Most teams interested in building instruments for Gemini have built other similar instruments that include a software component. Gemini acknowledges this, and our approach for software makes it easier to use builder software when appropriate for the task. But it is also true that not all builders have this background experience and in this case, it's necessary that Gemini provide more functionality and support in this situation.

2.1.3 Simplify the Instrument Integration Effort

The current model of instrument development delays much of the actual software test and integration effort to the final development days and commissioning of the instrument at the telescope. Then it is a rush to find out what things do and do not work and the debugging of software gets overloaded with testing basic instrument functionality. Software is inevitably late and often of poor quality. Cleaning things up gets relegated to later during operations.

This software approach defines a new process for developing software. Gemini and builders work together in an agile way during the instrument construction process to verify proper software operation. The product owner and builder cooperate to maximize results and be flexible to change to provide maximum value to Gemini.

2.1.4 Clarify Software Responsibilities

Building a Gemini instrument is a collaborative process between the builder and Gemini. Both groups need to understand what their responsibilities are, what their deliverables are, and when the deliverables are needed.

Each team needs to know when the other appears in their timeline. The GIAPI approach clarifies responsibilities for both the product owner and the builder.

2.1.5 Reduce Software Costs in Effort and Equipment

Software can often be a major cost in instruments and the increased cost is often because of non-technical issues. The selection of computing hardware has become more haphazard with recent instruments. Hardware and software choices are needed that can be supported and are appropriate for the tasks.

2.1.6 Increase Quality and Rigor

It is Gemini's goal is to provide more freedom and choice to the builder, but we are also asking that builders step up and provide increased levels of quality in the delivered software. Flexibility and reliability need to be considered from the initial stages. Iterative development, regular testing, and frequent releases must be an integral part of the software project development schedule (See section 14 in this document for more details). The viewpoint must be that the instrument is a Gemini facility instrument and the code is there to make the instrument work properly in the Gemini environment over the entire lifetime of the instrument. This needs to be the first priority when making software decisions. The focus is on the desired outcome—to produce a high-quality Gemini instrument that produces the desired science. Details on the development process are in Section 14.

2.1.7 Provide Timely and Appropriate Levels of Support

Gemini needs to focus its limited resources on providing the proper level and type of support to instrument builders. Not all builders need the same kind of support and not all instruments will have the same support requirements.

2.2 Splitting Builder and Gemini Responsibilities

The GIAPI was designed to allow the builder to progress with their design and development with as little dependency on Gemini as possible. This has been largely accomplished by providing a lightweight API that is targeted towards the specific types of information that Gemini needs from instruments.

2.2.1 Gemini Role

Gemini must provide the GIAPI integration library in the computer languages needed by the builders. Gemini must actively maintain and improve the library and its documentation in a way that focuses on the needs of Gemini and the builders.

Gemini views the instrument builder as a collaborator in the development of instrument software and our plan is to adopt the development processes described in this document in our support of builders and our development of GIAPI software. It is Gemini's responsibility to provide the builder with high quality information and software and to be responsive to builder needs. Gemini must provide the testing needed during the instrument development process that is required to ensure the builder software works properly when it arrives at the telescope.

2.2.2 Builder Role and Responsibilities

The builders' role is to provide the highest quality instrument software possible following the guidelines in this document and other related GIAPI documents. The instrument software that is constructed should understand and integrate with the Gemini system. For instance, the instrument builder should provide a high-quality ATEUI and provide proper FITS files to the GSDN (see sections in this document.) The builder must understand the GIAPI and how it differs from previous generations of instrument software at Gemini.

3 Gemini Instrument Background

This section provides a background in the terms and concepts used when discussing software for Gemini instruments. This background information is required for understanding subsequent sections that provide specific guidelines that focus on the hardware and software design of Gemini instrument software.

A typical Gemini instrument consists logically of three parts: the Components Controller (CC), Detector Controller (DC), and Instrument Sequencer (IS) as show in Figure 3 below. Circles indicate these three major components. The physical layout of the components is in shaded green; interactions with Gemini are shown in

yellow (see figure key). The example shows the Detector Controller separate from the computer, but that is not necessary.

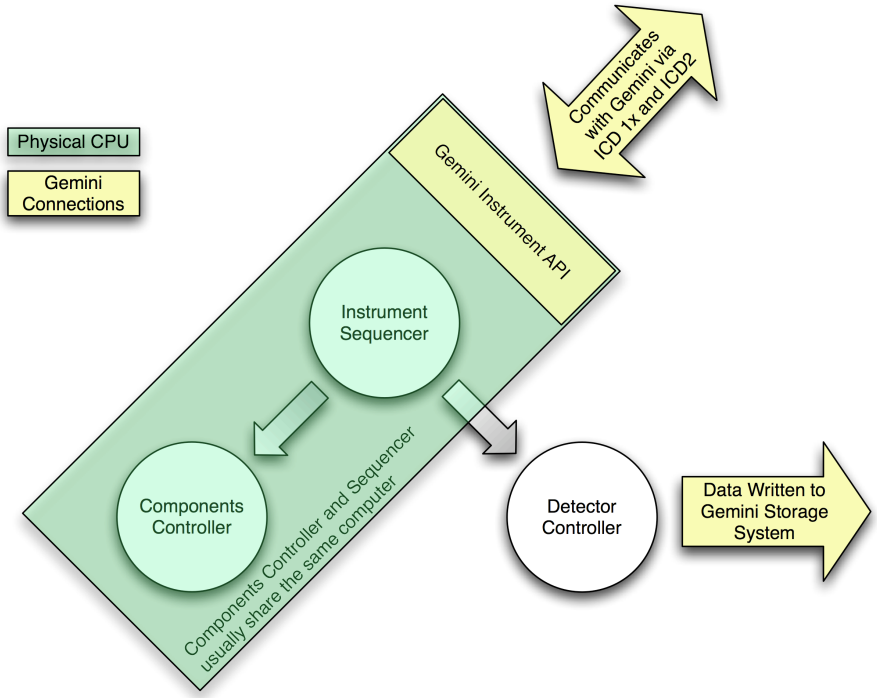


Figure 3: Major instrument software components

Each of these components performs a role that all instruments require (described below). They are logical components because it is not strictly necessary that the three software components are implemented as separate software entities—although it is usually desirable—and it is not necessary that they be provisioned across different computer systems.

Gemini Interface Control Documents (ICDs) govern the communication and interface between Gemini's four Principal Systems. The details of this communication are encapsulated in the GIAPI internals. The decisions about software within an instrument are primarily up to the builder. However, the internal design must be appropriate for the instrument and must meet the needs of Gemini as determined through the review process. The GIAPI can also provide some support for infrastructure within an instrument in some situations as covered later in this document.

3.1 Instrument Sequencer

The Instrument Sequencer has the responsibility of receiving configurations from the OCS and matching them by controlling the ICS subsystems and detector controller(s). The GIAPI provides functionality that allows the IS to receive commands/configurations and update Gemini with command completion information for actions caused by the commands/configurations.

- The Instrument Sequencer is the software component in the instrument that knows how to order activities within the instrument. The instrument must handle all internal sequencing details of the instrument (REQ-GIAPI-INS-01) and not expect that responsibility to be handled by the sequencing software in the OCS.

The instrument sequencer may have a significant role in providing status and health information and sequence command completion information to the OCS and other Gemini systems. In a hierarchical design, it is also possible that it shares and delegates this responsibility with the instrument DC and CC. How command completion information is produced in builder software is a builder design decision.

Section 10 covers GIAPI support for receiving commands and updating Gemini with completion information. Status support is provided in Section 9.

3.2 Components Controller

The Components Controller is charged with controlling the motors and other devices that are contained within the instrument. In response to a request from the Instrument Sequencer, it attempts to make the actual instrument configuration match the demanded configuration. The following are functions often associated with the Components Controller. It is not absolutely necessary that these functions are located within the CC, but the instrument software must handle them somewhere.

- Move the configuration of the instrument devices and motors from one safe state to another. (REQ-GIAPI-INS-02)
- Complex configurations resulting in multiple motions should be applied in parallel. (REQ-GIAPI-INS-03) If several different components undergo movement with one applied configuration, the devices should be moved in parallel to minimize the time needed to reconfigure the instrument.
- Handle any motion peculiarities within the instrument. (REQ-GIAPI-INS-04)

  Instruments with multiple filter wheels can have dependencies between wheels. For example, if wheel 1 is moved from A to G, wheel 2 must be also be moved to a blocking position such that no bright light ever hits the detector. Then following the conclusion of the filter 1 move, filter 2 must be moved back to its previous position. This kind of instrument-specific logic, if needed, should be embedded in the components **controller logic**. For instance, the GMOS instrument has two physical filter wheels that act as one to higher software levels.

  Some instruments (NICI, for instance) provide *modes*. When a mode is set it can control multiple motors and limit or change the options of other devices. An instrument components controller should support modes if it is warranted by the operations model and will allow simplified and less error prone instrument control.
- Be sufficiently knowledgeable about the current motor and device configuration such that unnecessary motions or movements are eliminated (GIAPI-REQ-INS-05). As a simple example, if a device is at Filter B and is then requested by the OCS through the instrument sequencer to go to Filter B, it must know that it is already there and complete immediately with no extra motor motions.

The CC also is responsible for control or status of the parts of the instrument not covered by the Detector Controller. Examples of such activities could include:

- Switching the calibration lamp(s) on and off.
- Sensing the position of switches on the instrument.
- Sensing the temperature of various locations within the instrument.
- Sensing and reporting error or health conditions within the instrument.

The CC often produces the majority of instrument status information and updates status during actions. Status is covered in Section 9.

3.3 Detector Controller

A traditional, conforming Gemini ICS design, as described in Gemini *ICD 7b, ICD Subsystem Interface* [11], views the Detector Controller (DC) as a separate software subsystem of the instrument, but this has not been entirely true in delivered instruments. Often the same instrument builder delivers the detector controller, and it is tightly coupled to the instrument architecture. In this case, the CC to DC software/hardware interface is a proprietary or instrument-specific interface.

The Detector Controller's job is to accept requests from the Instrument Sequencer and control the detector(s) to acquire science data.

The type of functionality usually within the detector controller includes:

- Providing control of hardware-specific detector parameters (voltages, etc.)
- Control detector waveforms.
- Possible control and synchronization with the Gemini secondary position when chopping.

- Can provide events and status information to Gemini through the GIAPI. For example, data acquisition started and data acquisition end events.
- Handle data from detectors and process it as needed.
- Forward the science data to Gemini through the GIAPI data protocols.

The GIAPI supports the delivery of data to Gemini as discussed in Section 12.3.

3.3.1 Internal Data Processor

The Detector Controller must provide any preprocessing of data that may be needed before forwarding to Gemini (REQ-GIAPI-INS-06). Examples of this kind of processing are:

- Co-adding many short exposures together.
- Joining one or more exposures together.
- Subtracting beams A and B for chopped data to make one sky-subtracted frame.
- Changing the units of the data.
- Rotating, flipping, detangling, reorganizing or otherwise processing the data to make sensible datasets.

Well-formed, easily viewable datasets must be passed to Gemini. The instrument should not rely on the Gemini OLDP pipeline to do this kind of processing. (REQ-GIAPI-INS-07)

4 GIAPI Functional Overview

This section provides an overview of functionality the GIAPI provides to support integration of the instrument software into the Gemini environment. The functionality is grouped into two different sections; each of which references topics that are more completely described in later sections.

New integration features will be added to the GIAPI as needed based on new instrument requirements and requests from builders.

4.1 Commands, Status and Science Data

Producing the correct science data is probably the most important task of the instrument software. The Gemini control system sends commands in the form of configurations to the instrument to specify how the instrument should take the data. The instrument tells Gemini how it is configured through status. Table 1 shows where to find these topics in the document.

Functionality	Description	Document Reference
Send instrument status to Gemini infrastructure	The instrument software notifies Gemini through the GIAPI that an item of status has been updated.	Section 9 on page 21.
Receive command configurations and provide action update information	The instrument will receive configurations from Gemini via the GIAPI. The instrument software must provide feedback on the success or failure of actions and status on progress.	Section 10 on page 27.
Create science and engineering datasets	Instruments produce some kind of data. The GIAPI defines the methods and protocols used by instruments to deliver datasets to Gemini.	Section 12.3 on page 46.

Associate additional files with datasets	Some instruments need to create additional information that must be associated with datasets but can't be included in the dataset headers. The GIAPI provides a method for associating information with science datasets.	Section 13.5 on page 52.

Table 1: Document references for commands, status, and science data creation

4.2 Instrument Integration with the Gemini Telescope Software and Hardware

The instrument software is part of a larger software system with *a lot of moving parts*[1] and depending upon the requirements of the instrument it is necessary to interact with that system in some way. Table 2 shows the interactions covered in this document with references.

Functionality	Description	Document Reference
Receive status information	The instrument needs a status item from one of the Gemini EPICS systems. An example of this is an instrument needing the current telescope azimuth or zenith angle.	Section 11.1 on page 41.
OIWFS Updates from AGCS	The instrument needs to get tracking demands for an on-board wavefront sensor from the telescope control system through the acquisition and guide unit.	Section 11.2 on page 43.
Provide wavefront corrections to the PCS	Some instruments with on-board wavefront sensors need to provide the Primary Control System with slowly varying corrections.	Section 11.3.1 on page 43.
Instrument needs to control the secondary mirror	Instruments that provide information to the Gemini Secondary Control System must interact with the Gemini synchro bus. GIAPI provides a device driver and software interface to allow this functionality. The GIAPI also specifies hardware requirements.	Section 11.5 on 44.
Instrument needs WCS information	Instruments need to compute World Coordinate System information for dataset headers using information from the Telescope Control System.	Section 11.4 on page 43.
Receive and use time that is synchronized with Gemini systems	The GIAPI provides the instrument builder with the capability of receiving time that is synchronized with Gemini systems.	Section 12.2 on page 45.

Table 2: GIAPI support for interacting with telescope systems

[1] A new Gemini programmer made the comment that it was miraculous that the telescope operated at all given the large number of moving parts during observing.

5 Computer Hardware Choices

A Gemini instrument usually consists of more than one computer. Current conforming Gemini instruments are based on at least one VME rack-based system with at least one PowerPC CPU board (or MVM167 for older systems). This choice was typical in the mid to late 1990's when the development of Gemini systems was underway, but it's an approach that is no longer needed in most situations, and its use can add to the complexity and cost of the software and hardware.

For new instruments, the choices were revised to offer options that included more convenient configurations and more modern hardware in addition to the embedded VME hardware option. During the conceptual design process it became clear that there was little interest in continuing to use a VME embedded system solution, and this hardware choice has now been dropped unless an application can be shown to require it. The kind of situation when this choice is appropriate is when code is very coupled to hardware control or has hard real-time requirements.

Builders are moving towards common, cost-effective hardware for their instruments and by targeting this platform we can minimize costs and focus our effort on the use of modern computing hardware. Gemini supports the use of common x86-based hardware to bridge the builder environment with the Gemini environment. The configuration of a new instrument computing system is shown in the next section followed by specific hardware recommendations.

5.1 High-Level Physical Layout of a Gemini Instrument

A description of how the software components are distributed across the set of processors and computers that makes up the instrument is the instrument's physical computing layout. As discussed in Section 3, the instrument consists of three logical software components: the Instrument Sequencer, the Components Controller, and the Detector Controller. This section discusses how the high-level components should be distributed and defines some terms used in the following discussions.

A major goal with Gemini instruments is to keep the number of computers and processors down to the minimum number needed to do the job. Computers should be used efficiently and there is not a single, magic number; the right number depends on the instrument. Arriving at the value for the number of computers is a design decision for the instrument software builder. Mimicking the organizational structure of the project is not always the best solution (see Conway's Law - http://en.wikipedia.org/wiki/Conway's_Law). A secondary goal is to isolate the instrument interaction with Gemini allowing a simpler design for the instrument.

To attain these goals the idea of a *Top-Level Computer* (TLC) is being introduced. The top-level instrument computer should be the system for the implementation of the following features and components:

- The primary system running builder code that integrates with the Gemini software interface to commands and status.
- Provides the Instrument Sequencer capability.
- Provides all or a major part of the instrument Components Controller functionality.

These components are co-hosted on the TLC in order to reduce the number of computers, which reduces software complexity for common operations, reduces command execution times, increases overall performance, and removes reliability issues that are increased by inter-computer communications.

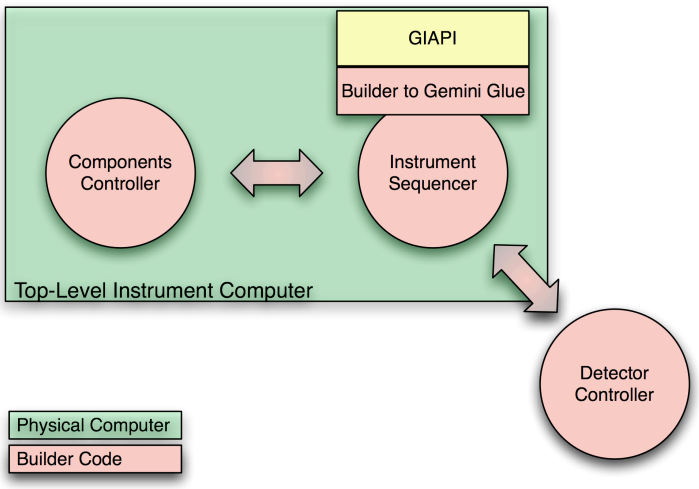


Figure 4: Preferred top-level component layout

The number and role for additional computers needed by an instrument to supplement the mandatory TLC are the concern of the builder. Hardware choices and physical layout for computer control of the instrument must be designed, documented and justified as part of the instrument design process (REQ-GIAPI-HW-09). Solutions that stray far from recommendations must be justified during reviews.

It is recommended that all computers in the instrument be as similar as possible. This reduces costs for everyone, long-term maintenance is easier, and purchasing spare parts is easier and less costly.

5.2 Hardware Choices

Gemini will eventually recommend specific hardware for the TLC, but it is currently too early in the development process to do that. Given the long system development time, it is preferable to delay hardware choices as long as possible. Recommended systems will be determined when it becomes necessary for the instrument to explicitly specify their equipment. This needn't be a roadblock. It's possible to estimate the cost of the hardware of the future based on today's costs. Computer equipment costs for this type of system are relatively constant in time.

The following are the requirements for TLC hardware. It is preferred that all instrument computers follow the TLC specifications unless there is a significant reason not to do so (REQ-GIAPI-HW-10). Distributing GIAPI functionality to other computers besides the TLC requires hardware that meets the same specifications as the TLC (REQ-GIAPI-HW-11). Gemini instruments are expensive as is telescope time. Instruments cannot run without the computers being operational, so it's important to use quality equipment. Note that these hardware requirements will change over time as available hardware changes. Table 3 summarizes the rest of the requirements presented in the *GIAPI Builder Requirements* [21] related to hardware.

Requirement ID	Requirement Name	Description
REQ-GIAPI-HW-01	CPU	System should use AMD or Intel 64-bit capable x86 CPU(s) using multiple processors and/or multi-cores as needed.[1]
REQ-GIAPI-HW-02	Redundant power supply	The computers should have redundant power supplies. The important requirement is that the computer not go down when one of the power supplies fails and that no human intervention is needed to switch to the backup.

REQ-GIAPI-HW-03	Redundant, accessible disk drives	Disk drives should be redundant allowing a RAID setup. Disks should be easily accessible from the front and allow hot-pluggable exchange without opening the chassis. Solid-state drives (SDD) such as the Flash memory-based drives should be considered for drives within the dome environment.
REQ-GIAPI-HW-04	Error correcting memory	Systems should use registered ECC memory.
REQ-GIAPI-HW-05	Add-in board support	Systems should support at least 2 full-height PCI-X or PCI Express slots.[1] Also see Section 5.2.4 below.
REQ-GIAPI-HW-06	Dual Ethernet ports	Systems should provide at least 2 Ethernet 10/100/1000 Ethernet NICs.[1]
REQ-GIAPI-HW-07	USB-2.0 ports	The system should provide at least one USB 2.0 port to allow booting from a USB-based memory device.[1]
REQ-GIAPI-HW-08	NEBS compatibility	Given the harsh enclosure environment and extreme conditions within the telescope enclosure NEBS Level 3 compatible or carrier-grade equipment is preferred unless systems are mounted in the computer room.
1. This specification may change over time to reflect the state of available hardware at a future date.		

Table 3: TLC hardware requirements

5.2.1 NEBS Compliance

The requirement for NEBS-compliant (REQ-GIAPI-HW-08) equipment is fairly restrictive; not too many manufacturers of equipment provide computers that satisfy this standard, but they do exist. For instance, Sun and HP both provide NEBS Level-3 equipment (often called *carrier-grade*). The requirement for x86 processors also restricts choices (Sun has only one x86-based NEBS server).

Providing equipment that satisfies the other requirements captures a large part of NEBS-3 compliance and is an acceptable approach. An instrument at Mauna Kea sees a harsher environment than Cerro Pachon. Hard drives, in particular, should be carefully considered when deploying on Mauna Kea (see 5.2.3 below.) Good air ventilation of computer equipment is critical on Mauna Kea.

5.2.2 Two Phase Hardware Planning

The instrument software development process can be lengthy and purchase of computing hardware too early in the process can lead to equipment that is obsolete and hard to spare before it is delivered. This problem can be alleviated with a hardware purchase plan that includes early purchase (if necessary) of development equipment followed by the purchase of final, to-be-delivered equipment late in the development phase.

This approach is tractable because the hardware is based on available equipment and processors. The Linux OS choice reduces risk as well. Minimizing the use of hardware that is resident directly in a computer bus helps minimize OS and hardware dependencies. The development hardware does not need to meet the stringent redundancy requirements of the final hardware reducing costs but still allowing prototyping and performance evaluation opportunities.

5.2.3 Disk Drive Considerations

Placing hard drives within the TLC (or other instrument computer) in the dome environment is challenging but has been done successfully in TReCS. There are other alternatives that can be considered as well. If a relatively small amount of storage (64GB at this time) is needed, then solid-state drives could be a good choice for consideration.

Another possibility is to place the instrument disk systems within the computer room connected by private Ethernet (for a NAS system) or optical fiber if higher performance is needed. Optical fibers are already available in the telescope infrastructure.

5.2.4 Computer Bus Considerations

The intent of the bus requirement is not to specify a specific bus choice; rather, it's to ensure there are 2 slots available using the commonly available bus.

The bus choice for the TLC is determined by market availability when the hardware decision for the instrument must be made. As of this document version, the most easily available x86-based hardware includes a mixture of PCI Express (PCIe) and/or PCI-X. If the hardware choice for an instrument must be made today, these two bus choices are acceptable. In many cases, the actual bus choice is irrelevant since there will be no instrument specific boards and device drivers.

Compact PCI (CPCI) is a different case. CPCI is not included in the easily available x86-based hardware we are targeting for the TLC. A CPCI-based system might have a role in some instrument since it is often targeted at high-speed industrial computing application and data acquisition. An example is the PXI National Instruments hardware.

This requirement specifies a minimum of 2 slots. These two slots are reserved for Gemini use—one for a possible fiber channel disk interface and one for possible time bus integration. If an instrument needs to install additional boards in the TLC, hardware with more than the minimum of 2 slots should be procured.

A specific instrument may not need the two Gemini features, in which case the number of Gemini reserved slots can be reduced during instrument design. (See Section 12.2 on time access and Section 13.2 on writing science data.)

5.3 Network Environment

The current Gemini network environment is based upon Gigabit Ethernet with the majority of the infrastructure based upon the 1-gigabit specification. It is expected that this environment will continue through the commissioning period of the instruments. More information on the telescope network environment is available from the Gemini Information Systems Group and will be included in the Builder Requirements document.

5.4 Network Storage System

The telescope summit environment provides instruments with a Network Attached Storage (NAS) system to be used for writing instrument datasets. The current NAS system is based on equipment from Network Appliance Incorporated (http://www.netapp.com) and provides multiple terabytes of storage space.

Builders can assume the NAS system will appear on their instrument as a mounted file system. The explicit method for attaching the storage system to the instrument computer may be NFS over Ethernet or a higher performance solution such as Fibre Channel or iSCSI. While a decision to use one of the higher performance solutions does not impact the software design it would require an additional hardware board in a computer system using the storage network.

6 Operating System Choices

The TLC computer runs the Linux operating system. At this time, the specified version for this type of application at Gemini is CentOS 6 or newer (reference to ICD) (REQ-GIAPI-OS-01). It makes sense to run the 64-bit OS on 64-bit CPUs, but it's not clear that creating applications as 64-bit applications is necessary. Builders should consider this decision during design.

This item indicates the OS for the TLC, but we recommend using the same OS on all the instrument computers just as we recommend using the same computer configuration as the TLC. Using the same OS is preferred because we have to support it after instrument builders are long gone. If there are good technical engineering reasons why CentOS shouldn't be used for all the computers, the arguments should be made during reviews.

This specification is likely to change given the rapid changes in Linux and hardware, but changes in Linux OS should not impact builders.

7 GIAPI Architectural Overview

This section describes the overall architecture of the GIAPI and demonstrates how it is deployed in typical instrument scenarios. After presenting the architecture, the document includes several sections that describe the support for specific aspects of interfacing an instrument to the Gemini system within the GIAPI environment.

The main part of the GIAPI environment consists of two pieces: the *Gemini Master Process* (GMP) and one or more implementations of the language-specific API or GIAPI glue (referred to as API glue). The Gemini Master Process is a Gemini-provided component that provides the bulk of the Gemini features and integration functionality. The language-specific glue is the API layer used by the builder to implement integration functionality. There are different versions of the glue for different programming languages. The glue API is implemented as a shared library in C++ and a JAR file in Java.

A simple instrument that shows the relationship between builder code, the Gemini Master Process, and the language-specific API glue is shown in Figure 5.

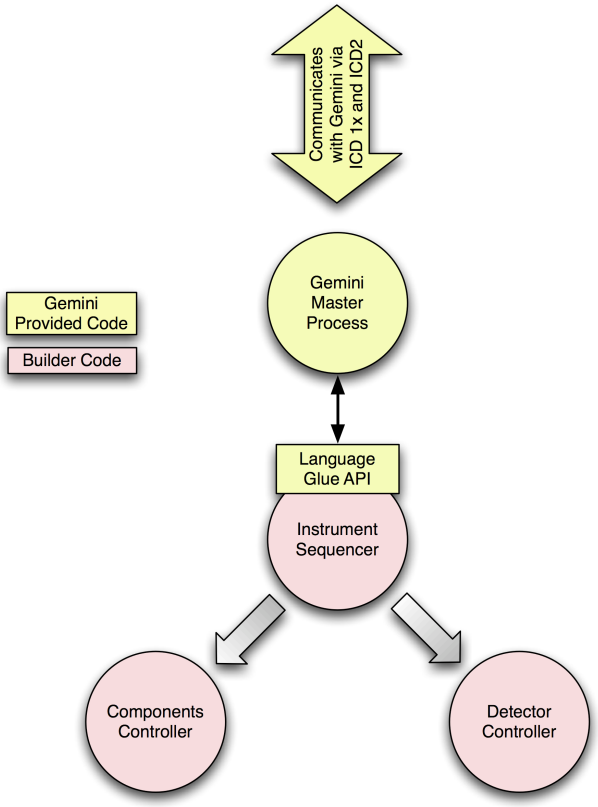


Figure 5: Simple instrument showing Gemini Master Process and language-specific API

This figure assumes the implementation of the instrument in the Linux OS consists of at least one process other than the GMP (note the instrument software cannot be combined with the GMP). The following are important, basic GIAPI usage points based on this figure:

- Note that the GMP encapsulates and handles the details of all communication with current Gemini software. For command and status purposes, the builder is only concerned with the details of the GIAPI as documented in the language-specific glue API ICDs.
- The GIAPI language-specific glue API library is directly included with any process that the builder creates that must communicate with Gemini. This figure is showing the Instrument Sequencer linked with the GIAPI. It is the builder's instrument architecture and design that determines which processes in the instrument must link with and use the glue API.
- The communication protocol between the language-specific glue API and the GMP is encapsulated within the GIAPI implementation.

Figure 5 is a very simple case. The following figure shows some more sophisticated options enabled by the GIAPI implementation.

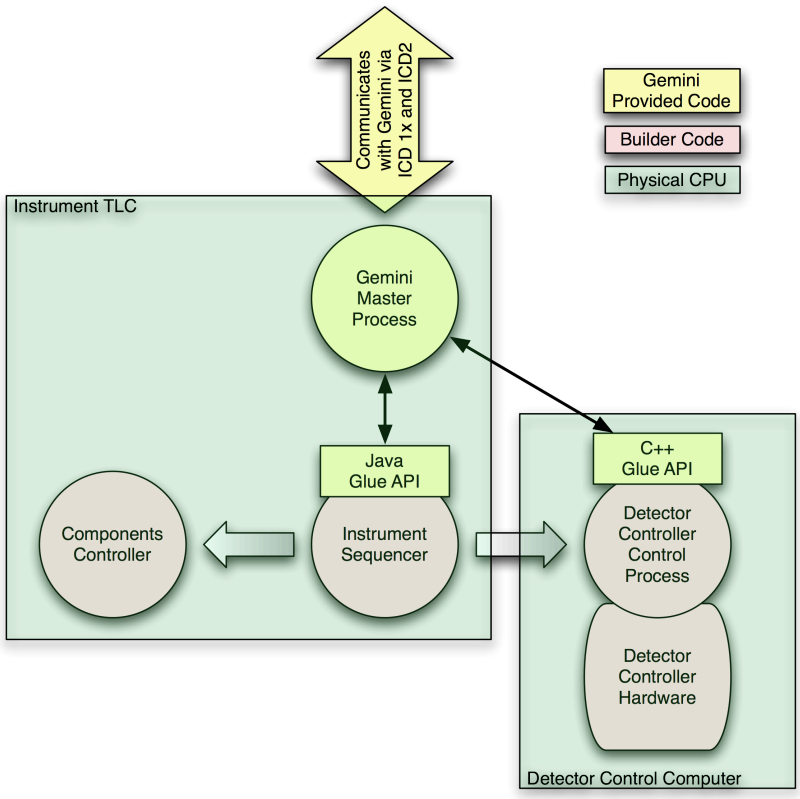


Figure 6: More sophisticated instrument using GIAPI

Figure 6 shows a more complicated instrument by adding in physical deployment of the instrument components on two computers. The figure assumes a proprietary instrument communication scheme, shown by the larger arrows, is in use between the IS and CC and the IS and DC. The IS sends the CC and DC commands via the private channel. The following are important basic GIAPI usage points based on this figure:

- In this hypothetical instrument all the motors are controlled via standalone Galil controllers that are accessed via a private instrument Ethernet network. The components controller and instrument sequencer are written in Java to speed development and simplify debugging. The IS uses the Java language glue API to interface with the GMP.
- The Detector Controller is deployed on a second computer that hosts a C++ process that interfaces directly with detector controller hardware that is installed directly on the computer bus. Because of the low-level hardware access, the process is written in C++ and uses the C++ glue API to interface with the GMP.

More than one process in the instrument can communicate with the GMP, and a single computer can host more than one process that communicates with the GMP. Processes that communicate with the GMP can be written in any of the supported glue languages. It is not required that all processes in the instrument or on the same computer system communicate with the GMP using the same language glue.

7.1 Implementation Overview

Although not needed to use the GIAPI, it is useful to understand the technology used in the GMP because it enables easier understanding of how the GIAPI and GMP work to provide the needed functionality.

The GMP is written in Java and relies on two standards: Java Message Service (JMS) and Open Systems Gateway Initiative (OSGi).

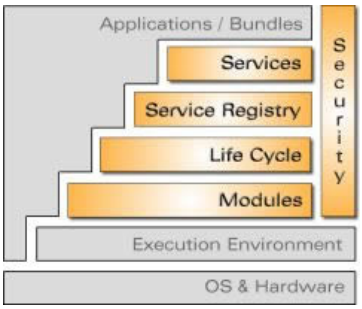
7.1.1 Java Message Service

The Java Message Service is a Java-based standard for reliable message-based communication. It is called a Message-Oriented-Middleware solution and is included as part of every J2EE distribution. JMS is a specification for a standardized API, and implementations of the specification must work the same way. The

JMS API provides a flexible, reliable service for asynchronous exchange of data and events throughout an enterprise. It provides a common set of messaging concepts and programming strategies that will be supported by all JMS technology-compliant messaging systems [1].

The JMS home page references 13 commercial implementations of JMS. There are 9 open source implementations of JMS listed at http://java-source.net/open-source/jms. Gemini is currently using the open source ActiveMQ [4] message bus implementation of JMS as the core of the GMP. ActiveMQ is a capable, well-supported package currently in the Apache incubator stage. It was chosen because of its performance and its support for a wide variety of languages and connectivity options. ActiveMQ does more than implement JMS; it is a message bus with JMS at its core. The home site [4] contains and abundance of information. Some sections in the remaining parts of this document will reference JMS features that are used to provide support for GIAPI functionality.

7.1.2 Open Systems Gateway Initiative Framework



The Open Systems Gateway Initiative (OSGi) Service Platform is a Java-based framework that enables loosely coupled services to dynamically discover each other and collaborate to form applications. The framework specifies an environment that allows multiple applications to run within a single JVM as services. It provides full lifecycle support to install, start, stop, update, and uninstall applications over the network without affecting other applications executing within the framework. Additionally, a set of common services is defined that provide many of the services needed in modern, component-oriented middleware.

The OSGi Alliance is a standards organization supported by over 30 large companies. These companies have been working together since 1999 and the OSGi specification is now available as revision 4 [5].

There are multiple commercial and open-source implementations of the revision 3 and 4 OSGi frameworks. The Eclipse tool platform is based on the Equinox R4 OSGi implementation [6]. Gemini is currently using the Apache Felix R4 implementation [8].

The big advantage of OSGi for GIAPI is the ability to construct the software as a collection of highly cohesive, loosely coupled components (plugins) that can be modified with reduced risks of side effects.

7.1.3 GIAPI Implementation Using JMS and OSGi Framework

With this background, this section provides more specific information on the implementation of the GIAPI based upon the OSGi Framework and JMS terminology.

The Gemini Master Process is an OSGi application consisting of a number of bundles and services that will be used by Gemini. The JMS Provider (the central message router in JMS-speak) is one of the bundles in the OSGi application. Other bundles will provide features Gemini wants such as EPICS access to the instrument through the Java Channel Access Server that was recently completed by CosyLab (under contract from Gemini). The instrument builder does not need these additional features. Figure 7 is another view of Figure 6 showing the GMP as an OSGi application.

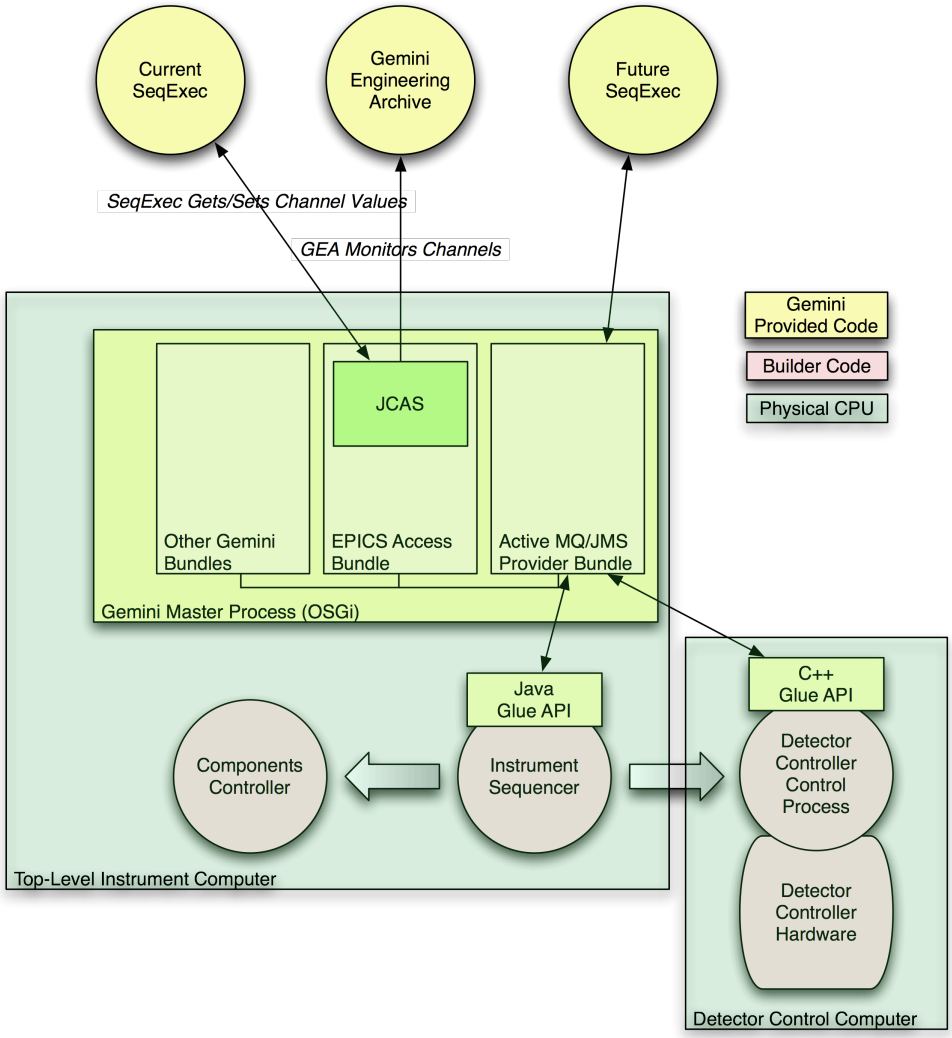


Figure 7: Expanded view of the Gemini Master Process

This figure shows connections to Gemini infrastructure external to the instrument and expands the Gemini Master Process to show some of the contained OSGi bundles. As in the previous figure, the instrument consists of two computers: the TLC and a detector control computer. The OSGi-based GMP executes only on the TLC. The instrument code connects with the GMP through the Java Glue API from the Java-based IS and the C++ Glue API from the DC process.

One OSGi bundle in the GMP includes the ActiveMQ JMS Provider. This bundle is one endpoint for all instrument communication with Gemini. Gemini is supporting a small subset of the connectivity options provided by ActiveMQ. The Java and C++ Glue APIs wrap the ActiveMQ language APIs to provide an instrument-specific API that directly supports the things instrument software needs to do to interface with Gemini (a discussed in upcoming sections.) Use of the ActiveMQ or JMS interfaces outside of the Gemini APIs is not allowed without specific permission from Gemini.

The bundles and services in the GMP share the same JVM but are isolated by the OSGi classloader infrastructure. The line at the bottom of the bundles is meant to show that bundles communicate with each other through service interfaces. For instance, information that is being published to the Gemini Engineering Archive is passed from the Provider bundle to the EPICS Access Bundle where it is published over EPICS Channel Access. This happens, for instance, as a consequence of the builder publishing a status item.

The EPICS Access Bundle also provides connectivity to the current generation Gemini SeqExec application. The figure also shows a future SeqExec that communicates with the instrument through the Provider Bundle rather than the EPICS Access Bundle. This is a Gemini detail that should not influence the builder software design (see the testing discussion in Section 15).

7.1.4 Performance Integration Issues and Risks

The approach taken with GIAPI requires Gemini infrastructure code and builder-produced instrument code to execute concurrently on the same computer. Ultimately and obviously, the performance of the overall instrument in the Gemini environment relies on the performance of all of the parts. The product being built is a Gemini instrument, not an instrument that happens to run at Gemini; building instrument software that works with the GIAPI is an engineering priority. TLC hardware should be sized considering the GMP as well as the instrument code.

It's good engineering to be interested in how the GMP will influence the performance of the top-level computer and minimizing the GMP impact on performance is definitely a high priority for GIAPI development. Possible performance issues in both the instrument code and the GIAPI need to be tracked and removed.

The development plan allows us to test for and address performance issues early (See Section 14.) The instrument software development plan should emphasize early delivery of software that demonstrates end-to-end operation so that together we have plenty of opportunities to test the entire system well before final integration. Iterative development and early delivery reduces risks like this.

7.2 Language Glue Choices

The GIAPI target two Glue API languages: C++ and Java. All GIAPI functionality will be supported in each of the Glue API packages. The goal is to provide an API in each of the language that looks very similar to ease maintenance and allow builder programmers to more easily use multiple languages as appropriate. Currently, the C++ version is being used by the GPI project.

Builders should use good engineering judgment when choosing the appropriate language glue for an instrument or an instrument subsystem. GIAPI supports both C++ and Java in the same instrument and mixing them in an instrument is reasonable, but Gemini suggests avoiding mixing languages in an instrument when one language is dominant.

7.2.1 The Need for Standardization

The GIAPI approach requires Gemini-produced libraries be linked with builder code. For this to be successful there must be agreement on a number of configuration choices.

The GMP and Java language glue are written in Java and are as portable as the Java Virtual Machine. Sun provides a Linux JVM. The current version is Java 6 (1.6).

The C++ language glue library and other C-language libraries are more of an issue. To ensure that the C++ and C code is reasonably portable we are standardizing the OS release as well as the compilers that should be used. No C++ or C features outside of the respective ANSI standards will be used. The specific compilers and operating system options used in the C++ language glue is documented in the corresponding ICD [19].

7.3 GIAPI Infrastructure Enhancements

The idea that instrument builders have a pre-existing code base is one of the fundamental assumptions and hopeful cost reducers of these changes. However, early software studies and reviews indicated that not all instrument builders are using libraries of tested builder code, and it's not in Gemini's best interests to fund the creation of new instrument infrastructure efforts by builders from scratch.

For this reason we have enabled GIAPI enhancements that allow for an adequate instrument infrastructure based upon the GIAPI. These features will be described in the upcoming sections and noted by a red enhancement marker like this: E

7.3.1 Use of Enhancements

Any builder can use the GIAPI enhancements but we require that if the instrument uses the enhancements, they must commit to an instrument infrastructure based on the GIAPI enhancements and not use a mixture of builder-supplied and enhancement-based infrastructure. The use of the GIAPI enhancements must be based agreed to by Gemini instrumentation management since it increases the role of Gemini software in the instrument.

8 Library Usage

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

8.1 Naming of items in the GIAPI

Every status or command item that passes between the instrument and Gemini must have a unique name within the entire Gemini control system following the conventions defined in this section (REQ-GIAPI-USE-01). Figure 8 shows the naming convention for all GIAPI item names.

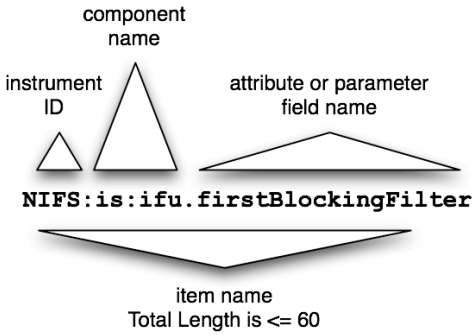


Figure 8: GIAPI item naming

The item name consists of the instrument ID followed by a component name and a field name. The field portion of the item name is separated from the component name by a period. All other parts of the item name are separated by a colon. The total length of the item name must be 60 characters or less. This length limit is determined by a limit in EPICS (version 3.14 and later).

The *instrument ID* defines a namespace for the instrument names and ensures uniqueness at the system-wide level. The instrument ID is part of every item in the instrument GIAPI interface. Within the scope of the namespace it is the builder's job to ensure that all GIAPI names are unique and meaningful. The instrument ID, shown in Figure 8 as NIFS, is usually three or four capital letters long and is an abridgement for the full instrument name. The builder and Gemini must agree on an instrument ID and document it in the instrument's ICD 16 public interface definition document.

Following the instrument ID is a *component name* that can be several names separated by colons. The component name can be used to partition the instrument into logical subsystems. In Figure 8 the component name is is:ifu.

The end of the component name and the start of the field name is indicated by a period. When discussing status values in the GIAPI, the field is called an attribute; with commands it is called a parameter. In Figure 8 the field name is firstBlockingFilter.

The status and command parameter and attribute names are part of the public interface of the instrument and are agreed upon during the instrument design phase to ensure that the correct instrument information is available. (REQ-GIAPI-USE-02)

8.1.1 Component Name Options

If the instrument architecture follows the Gemini approach with an instrument sequencer, components controller, and detector controller, item names can reflect the source or destination of the item in the instrument using the following convention in use by current EPICS instruments. The convention is to use CC, IS, or DC as a component at name separated from the field by a period. This is a convention, not a requirement. Examples are shown in Table 4.

Status Source	Example
Components Controller	INST:CC.filter1
Instrument Sequencer	INST:IS.countdown
Detector Controller	INST:DC.exposureTime

Table 4: Item name conventions with typical instrument components

The component name can be used to create additional name space when obvious field names would result in redundancy. For instance, an instrument might have several filter wheels that require status be published.

Another approach is to create more descriptive field names (e.g. INST:CC:nd.position or INST:CC.ndPosition). The following table shows possible alternatives using status as an example.

Example	Description
INST:filter	The instrument has so few status items that a component name is not needed.
INST:CC:IFU.order	The instrument has a special namespace for IFU attributes within the components controller.
INST:AOC.exposureTime	The instrument has a subsystem that is not one of the common Gemini instrument subsystems.

Figure 9: Demonstrating other component name approaches

More sophisticated schemes for component names are possible when the instrument has many subsystems, but complex names should be created only when it makes the interface clearer. This kind of decision should be discussed with Gemini. A single naming approach should be adopted uniformly within the TLC.

9 Providing Status to Gemini

The instantaneous state of the instrument is described by the values of all of its status items. This section describes how an instrument provides status information to Gemini using the GIAPI. The section starts off by providing some of the information from IG and then provides specific scenarios and requirements.

9.1 Uses for Status

An instrument must provide information describing its state and health to Gemini through the GIAPI (REQ-GIAPI-STA-01). The primary status information client is the OCS principal system that uses the information in the following ways:

- It coordinates the inclusion of status data in the science data headers,
- It updates the Science Program record with information that is part of the system status,
- It keeps the observer and operator screens up to date,
- It optionally can permanently store status items in the Gemini Engineering Archive.

An instrument may also need to receive certain status items from the Gemini environment. An example would be an instrument with a flexure adjustment that is dependent on the altitude of the telescope. This subject is covered in Section 11.1.

9.1.1 Status System Performance

The performance target for status publishing through the GIAPI is a minimum of one thousand status updates/second with a goal of 2500 status updates/second. Given current instruments, this is a reasonable target when limiting status to items of interest to science users. Builders should point out any problems with this maximum update rate. The ATEUI is the user interface that handles lower-level status items or values of interest to engineers.

9.2 Status Items

GIAPI hides many of the details of ICD 2, the document that describes the role of status in the Gemini control system. Some status-related definitions from ICD 2 remain relevant, and these are described in the next few paragraphs.

Status is made up of the values describing the instrument's state that an instrument makes available to the other hardware and software systems. A component's state includes the attributes related to the purpose of the

component but will also include system related status attributes such as its operational state (e.g., heartbeat, health).

A single instance of status is called a *status item* in the GIAPI. The most important information in a status item is its name and its current value. In the GIAPI a status item is an object in the glue language that contains the following things.

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.	char, string, boolean, float, double, integer, long[2]

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.

2. The set of data types may need to be reduced to something that is compatible between Java and C++. This set can be discussed if needed.

Table 5: Status item members

9.2.1 What Values Should be Published?

The intention is that values of interest to science users of the instrument be published through the GIAPI. For example, if a filter wheel goes from position A to D, it should publish the values B, C, and D when it attains the final position. The ATEUI will display engineering items that may not be of interest to science users.

If the instrument knows where a filter wheel is at all times, it should also publish intermediate positions during motions. In the previous example, the instrument should publish values A-B, B, B-C, C, C-D, D.

When the instrument does not know the position of the device it should publish the value "unknown".

In some cases the position can be posted as an alarm status item. A case where this might be reasonable is a power on where the device needs to be initialized before its position is known. These kinds of choices should be documented.

9.3 Status Item Alarm Conditions

An alarm is the asynchronous notification of an occurrence in the control system that is critical or important to proper operation. For example, an alarm should occur if a power supply fails within the instrument. The alarms are often not directly related to the observing activity but can have a profound effect on observing. In Gemini (through the EPICS infrastructure) a status item can include an alarm. An alarm has a *severity* and *cause*. (In EPICS the *cause* is called *status*, but using status here is confusing.)

Alarms can have a severity of *warning* or *failure*. A *warning* is an alarm that subsystems see as important and should be brought to the attention of the users. A component should be able to continue working in spite of any events that cause warnings. Warnings can be acknowledged by users allowing operations to continue. A *failed* severity indicates that a component is in a state that will not allow it to continue operations. The user or operator must solve the problem before continuing operations. Failed alarms do not go away until fixed. Users are typically made aware of alarms through the instrument user interface.

Alarms can also have a description of the alarm called *alarm cause*. The alarm cause is an enumerated type that describes the reason for the alarm. A *message* field is included to allow the instrument to post a descriptive message describing the alarm in a way that would be useful to a user. For example, if the instrument detector temperature is increasing above the workable limit, an alarm status item could be posted with severity of failure, cause HIHI, and a message stating the problem.

The alarm cause has a few pre-determined values for the common cases of a value high, high out of range, low and low out of range (similar to EPICS). There is also a possible value of other that requires a message to describe the cause. Table 6 summarizes this information.

Status Item Member	Description	Values
alarm severity	Severity indicates whether or not the alarm is catastrophic	ALARM_OK ALARM_WARNING ALARM_FAILURE
alarm 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 6: Alarm status members

Instruments must keep track of alarm conditions in the instrument and make that information available by *raising an alarm* when needed (REQ-GIAPI-STA-02). This means posting an update to the status item with the alarm severity and cause set. When an alarm does not make sense for a status item, it is not necessary to set it or track it.

9.4 Status Items and Instrument Health

The instrument must provide a pre-defined kind of status called *health* (REQ-GIAPI-STA-03). Health is an easily examined indication of the overall operational status of the instrument or a part of the instrument. The value of the instrument health can be: **good**, **warning**, and **bad**. The primary use of instrument health is as top-level status information for observers and users of the instrument. It is usually presented on instrument observing status displays.

Health is defined to be a component's well being as determined by the component itself. Health can have one of three values: *good* means *I am normal*, everything is okay; *warning* means *I'm operating but not normally*; and *bad*, which means *I'm not operational*. Health should be viewed as a predefined mandatory type of status.

It is up to the instrument developers to determine when their instrument is in each health state. Health must be kept up to date due to any changes in status items (REQ-GIAPI-STA-04). The typical implementation is one where individual subsystems or devices each define a health value and the instrument health value is recursively defined in terms of the children values. This is shown in Figure 10 where the overall system health is bad because the health of Device 1 in Sub-System 1 is bad.

There can obviously be a relationship between health, status, and alarms. For example, if the detector warms up such that it will impact observing, the instrument posts a temperature alarm status update with the out of range value and severity set to *fail*. This causes an alarm. In this case the health of the instrument should go to *bad* to indicate to users that the instrument is no longer operational until the temperature problem is corrected. The builder must consider and document the interrelationships of important status items (REQ-GIAPI-STA-05).

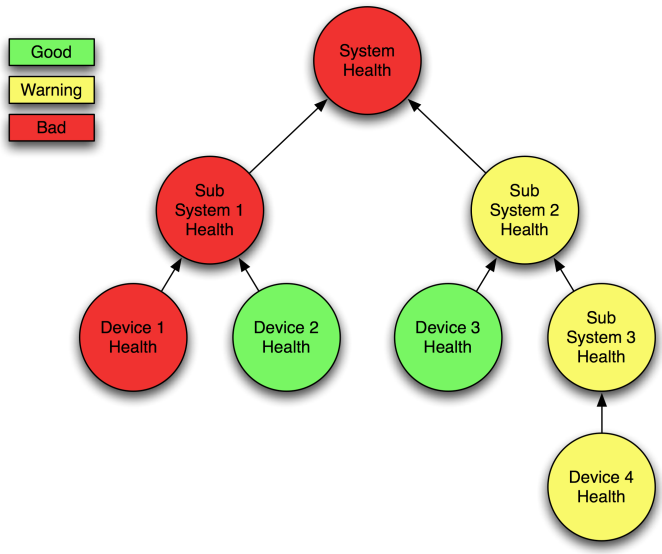


Figure 10: Recursively defined instrument system heath

It is not necessary that an instrument implement a health tree as in Figure 10. This is only an example of how health could be implemented and has been implemented in some existing Gemini instruments. GIAPI only requires that the instrument provide a single overall health that reports instrument health correctly (REQ-GIAPI-STA-03).

9.5 How To Use Status in the GIAPI

This section describes how status is described in the GIAPI. For all language glue APIs, the GIAPI provides the following capabilities for providing status information to Gemini:

- Publish (also called post) one or more status items to Gemini – this capability allows the builder to send one Status Item or a list of Status Items to Gemini via methods in the GIAPI.
- Publish one or more status items privately within the instrument.
- Subscribe and unsubscribe to private status items that flow only within the instrument.

9.5.1 Builder Responsibilities

The following are the builder's responsibilities with regard to status. All this responsibilities are tied to proper implementation of the ICD 2 publish/subscribe status model.

- Builder code should only post updates to status items that change (REQ-GIAPI-STA-06). This means do not burden the system with a set of updates that are not for changed status items. The GIAPI assumes the builder has only published changed status items.
- Builder code should post updates to status items when the values change. The builder software must meet status performance specifications when posting changes (see 9.5.5).
- The instrument must publish at least one health value known as INST:health (REQ-GIAPI-STA-03), where INST is the agreed upon unique name of the instrument. Other health values for subsystems can be published as well, but are not required.

9.5.2 Status Item Naming

Status items must have a unique name within the Gemini control system following the GIAPI convention in section 8.1. Figure 11 shows this for a typical status name. The field for a status item is called an attribute in earlier ICDs.

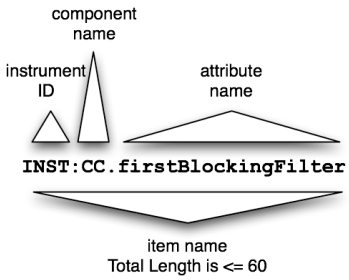


Figure 11: Status names

The status item name begins with the instrument ID, shown in Figure 11 as INST. Following the instrument ID is a component name that can be several names separated by colons. The component name can be used to partition the instrument status into logical subsystems. The end of the component name and the start of the attribute name is indicated by a period. In Figure 11 the component name is CC and the attribute name is firstBlockingFilter. Section 8.1 provides some guidance for creating component names for status items.

9.5.3 Public and Private

A status item can be *public* or *private*. A public status item is passed through the infrastructure to Gemini when updated. A private status item is only published within the instrument. Each glue API provides a separate method for publishing public or private status items (i.e. the public/private determination is made by a method call, not by something in the status item object.)

9.5.4 Private Status and Private Status Subscribing E

The GMP is based on JMS; therefore, it is easy to provide a way to subscribe to status items as well as to publish status items. Adding this capability allows an instrument to have a status infrastructure within the instrument with very little effort. The private status concept has been introduced to allow status to flow within the instrument but not to Gemini. Examples are shown in Figure 12.

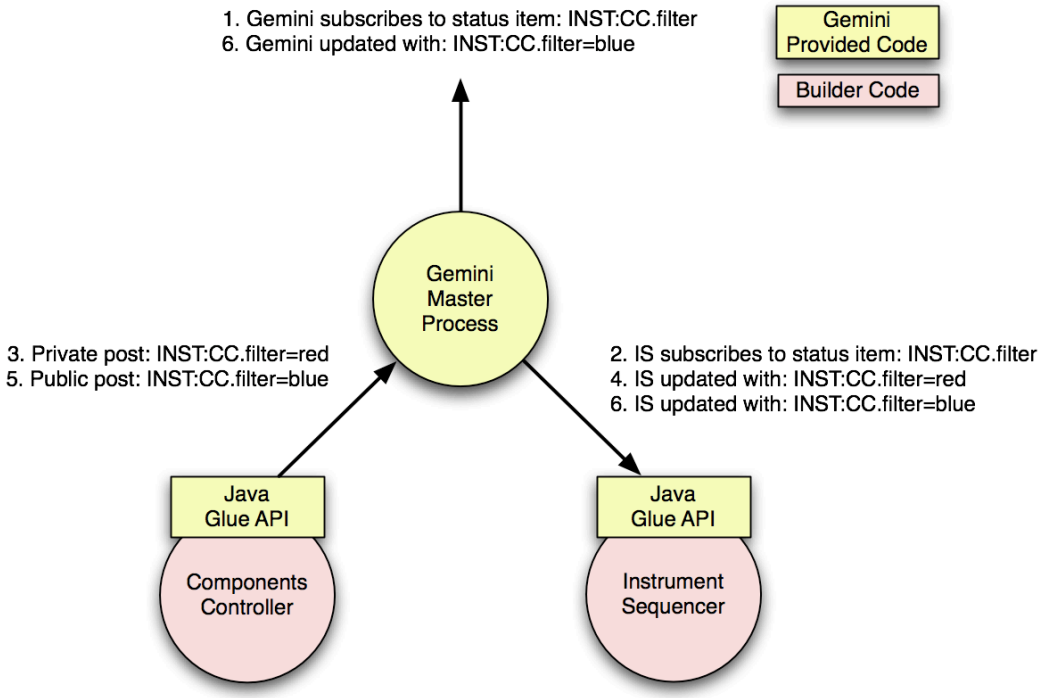


Figure 12: Demonstration of posting public and private status

Follow the numbers to see the order of activities in this example. First Gemini and the IS subscribe to the INST:CC.filter status item (1,2). Then CC does a private post of INST:CC.filter with the value red (3). The IS has subscribed to this status item and receives the update but Gemini does not (4). Then the CC posts another

update with the value blue (5). But this time the status item is posted publicly and both IS and Gemini receive the updated status item (6). Here are the important points regarding public and private status.

- Code must call a subscribe method to receive status item updates. The `subscribeInternalStatus` method is part of the GIAPI infrastructure enhancements.
- Status that is publicly posted travels to Gemini. This is done using a public post status method.
- Status that is privately posted travels only to clients that have subscribed to the status item within the instrument and not to Gemini.

9.5.5 Timely Status Updates

Proper operation of the Gemini control system depends on status items visible to Gemini be up to date at all times. If there is a delay in publishing status items, there is always a window where the status of the instrument does not properly represent the true state of the instrument. If it is large, this can potentially lead to user confusion or errors in the dataset headers. It is also possible that status subscribers will miss changes to the internal state because multiple updates can occur more quickly than the publishing period. Additionally, status must be delivered to Gemini in the order they were produced. This can be a problem with the polled approach. Like similar synchronization problems, effects due to failures in this area are difficult to locate and costly to correct late in the project when they are discovered.

To handle this need within GIAPI there are builder requirements defined that are traceable directly from the Gemini ICDs (p. 9, ICD2) and Gemini's use of EPICS. The publish-subscribe part is handled by the GIAPI, but the builder must ensure that the call to the GIAPI is made according to the requirements. The need to publish status items immediately is stated in Guidelines for Developing Gemini Instrument Software (Section 3.2.4, p.12 and Section 3.2.7, p. 13).

The internal implementation of status is a builder responsibility, but whatever implementation is chosen, it must meet GIAPI performance specifications.

9.5.5.1 Publishing Latency

The published status must always match the internal state of the instrument. The instrument must publish status values to the GIAPI within 300 ms of being changed (REQ-GIAPI-STA-07). Any larger would be noticed by observers and the window discussed above would be too large.

9.5.5.2 Updates While Changing

A status item that is changing continuously over several seconds should be updated at least every 1000 ms to and no faster than once every 500 ms to ensure the user is kept up to date (REQ-GIAPI-STA-08).

9.5.5.3 Lost Updates

No status value changes can be lost or missed (REQ-GIAPI-STA-09). For instance, in the situation where a filter goes from A to D, only publishing the final position D is not acceptable.

9.5.5.4 Ordering

Status values must always be published in the order in which they occur (REQ-GIAPI-STA-10). For instance, in the situation where a filter goes from A to D, status must be published as A, B, C, D, not A, C, B, D or some other order.

9.5.6 Scenarios

The following scenarios show how the instrument should behave in specific status scenarios.

9.5.6.1 Publish Status

Instruments need to publish changed status values as soon as they change. Consider the example of a filter wheel with 5 positions named A, B, C, D and E and an absolute encoder. The current position of the filter is given by the status item named INST:CC.filter with current value A. When requested to go to position E, the filter device will move to position E through positions A-B, B, B-C, C, C-D, D, D-E finally reaching E.

As the filter passes through each of the positions, the instrument software must publish a new status value for attribute GPI:CC.filter. Users want to know what is happening at all times. This is critical to the proper operation of Gemini and is basic to the publish/subscribe status model of ICD 2.

9.5.6.2 Alarm Scenario

Alarms are implemented as alarm status items.

Consider the previous example filter wheel has a soft and hard limit on either end of its travel. It can continue operation when in the soft limit but it is a failure if it reaches a hard limit. This is a common situation. The hardware presents the following values (ignoring intermediate positions): HL, SL, A, B, C, D, E, SH, HH. HL and SL mean Hard Low and Soft Low, respectively. SH and HH mean Soft High and Hard High, respectively. Now, because of a failure, when the filter goes from A to E, it passes E and reaches an undefined location in the SH limit. The INST:CC.filter alarm status:severity is set to SH:warning. If the device ignored the soft limit entirely and continued, it would report INST:CC.filter with an alarm status:severity to HH:failure. The value for INST:CC.filter in this situation is instrument-dependent.

9.5.6.3 Health

Health is implemented as an alarm status item with a value of good, warning, or bad. The message is used to describe a health problem.

Health is often tightly coupled to the alarm state of status values. In the previous example, suppose the instrument supports two health status values: INST:health, the health of the entire instrument, and INST:CC.health, the Component Controller health. INST:health depends directly on the health of INST:CC.health and INST:DC.health. When INST:CC.filter goes to the soft limit of SH:warning, INST:CC:health should show a value of *warning*, and this causes INST:health to also be set to *warning*. The continued movement of the filter could cause the health to be set to *bad* if the device can not be fixed, but typically another motion or datum will fix the problem.

9.6 GIAPI Functionality Summary

The following are GIAPI method calls that support the use of status.

`postStatus` – This method is called with a list of one or more status items (or the equivalent such as names of status items). When the method returns, the status items will have been sent to Gemini.

9.6.1 GIAPI Infrastructure Enhancement Functionality Summary

`subscribeInternalStatus` – This method is called with the name of a status item that should receive updates. It takes as arguments the status item name and a method that will be called when the item is updated. The argument of the update method is the StatusItem object.

`unsubscribeInternalStatus` – This method is called with the name of a status item currently receiving updates. It takes as arguments the status item name

`postInternalStatus` – This method is called with a list of one or more status items (or the equivalent such as names of status items). When the method returns, the status items will have been sent to any internal subscribers.

9.6.2 Usage Patterns

This section will give some examples of using status in typical instrument situations. This section will be added in a future version.

10 Receiving Sequence Commands from Gemini

The central idea of commands at Gemini and in the GIAPI is that all instruments respond to the same, small set of mandatory commands called Sequence Commands. Higher-level Gemini sequencing software is written to use only this small set of commands with configurations as opaque arguments; therefore all instruments need to accept sequence commands.

This section provides terminology and background information related to sequence commands and then describes command interfaces in the GIAPI environment. Builder requirements and scenarios for using sequence commands are included throughout the section.

10.1 Command Definitions

GIAPI hides many of the details of ICD 1x, the document that describes the model and protocol for commands in the Gemini control system. Some command-related definitions remain relevant.

Sequence Command – All command communication between high-level OCS software and the other principal systems is based upon a small set of abstract commands called *sequence commands*. Abstract means each command has a defined general meaning that is interpreted and implemented in a way that is appropriate for the system receiving the sequence command. The sequence commands are defined below in section 10.3.

Configurations and Parameters – The OCS commands an instrument by demanding that the instrument match a specified configuration. The configuration is made up of parameters. Each parameter has a name and a value. This activity is called *applying a configuration*.

Command Completion – A commanded system tracks the progress of actions started when a configuration is applied and publishes the status of executing actions. This allows the commanding system to synchronize its activities to the completion of actions in other systems.

Sender – The application that is sending the sequence command to the instrument.

Receiver – Typically the instrument or the component within the instrument receiving the sequence command. The receiver may also be the code within the instrument that handles the response to a specific sequence command.

10.2 Action Command Model

The ICD 1x documents describe a model for command execution called the Action Command Model or ACM that is the basis of communication between systems at Gemini. This abstraction has its roots in EPICS where the result of writing to the field of an IOC (Input/Output Crate) record triggers *actions* in the IOC, and this behavior defines the Action Command Model. The basic behavior of ACM is supported in the GIAPI, but with some slight improvements that are possible because the platform is no longer EPICS/VME.

It's important to understand the phases of the execution of a sequence command and the ACM terms that are tied to the GIAPI methods. Execution of a sequence command occurs in three phases.

Acceptance – In the ACM a sequence command is sent to a destination process and the message and its contents are immediately accepted or refused by the receiver. Acceptance means the actions associated with the request will be started while refusal indicates the contrary. Refusal of a request may occur, for instance, if a command's arguments are invalid or the related actions are already in progress and cannot be altered.

Busy – When a sequence command has been accepted the sender can assume that the actions have started successfully; any associated actions are said to be *busy*.

Command Completion – The receiver of the command must indicate to any interested parties when any actions associated with a request are completed. Actions can be completed successfully or with an error. This is called *command completion*. A sequence command is not completed until the sender receives command completion information for the requested actions.

It is wrong to accept a sequence command and then produce an error using command completion that should have been handled during acceptance.

GIAPI allows a receiver to indicate that a request is completed immediately at the end of the acceptance phase. This is useful for actions that do not take measurable time to execute (See section 10.6.)

10.3 Sequence Commands

Sequence commands can be grouped together by function as explained in the following four sections. In all text that follows, names of sequence commands are indicated in bold. The emboldened word **datum** means the datum sequence command.

10.3.1 Instrument Initialization, Setup and Shutdown

Table 7 contains sequence commands related to instrument initialization and shutdown. These commands are typically used by SSAs and observers when the instrument is setup at the beginning of an observing session, at the conclusion of observing, or as a recovery procedure when a failure or problem is encountered during observing.

Sequence	Arguments	Definition/Action

Command		
test		A system should assume it has just been switched on and perform its self-test suite for software and hardware systems to check that it is healthy.
reboot	GMP/gmp (optional) restart (optional)	This command causes the system to reboot (and restart EPICS if it is an EPICS-based system). After rebooting, this system should perform the same actions as the **init** command (see below). The restart argument is only used within the GIAPI and is ignored on an EPICS system. For Linux-based instruments the GMP will handle **reboot** as discussed below in section 10.3.2.
init		The system should restore itself to an initial condition, reloading initialization data.
datum		A datum command causes the system to place its mechanisms into a state (or position) where they can be moved reliably.
park		The instrument should adopt an internal configuration in which it can be safely switched off. Following a successful **park**, an instrument should be ready to be powered off.

Table 7: Setup and maintenance Sequence Commands

The following points are relevant to these sequence commands.

- **Test** implies that an instrument must have a set of self-tests that are documented in the public interface (REQ-GIAPI-SC-01).
- **Init** and **datum** are not the same. **Datum** initializes only the devices, not the rest of the software. **Datum** moves the devices to their *datum* positions. As an example, a **datum** for a device with only an incremental encoder could involve some special movements to set the encoder and device at a known location. The known location is often not a useful or meaningful device position for users (for example, a datum'ed filter might be left at a datum switch position rather than the "dark" position.)
- **Init** can also move devices. The difference is that **init** moves devices to their initialized positions assuming that all devices are properly datumed. There is no assumption that **init** will do the actions of **datum**. A device's position after **init** may or may not be the same position as its **datum** position (the position of the device after a **datum** sequence command). For example, **init** would likely move a filter wheel to the "dark" position. **Datum** may or may not leave the filter in the dark position. **Init** also initializes things that are not related to hardware (**datum** is only hardware). It reloads configuration information like lookup tables and initializes internal state.
- After an **init** successfully completes, the instrument should be in an initialized state and be ready for observing. This definition is more specific than what was stated in the past, but it is what we want
- Builders need to document in the instrument's public interface what devices and systems are affected by each of the setup and maintenance sequence commands (REQ-GIAPI-SC-02).

10.3.2 Handling Reboot on Linux

Handling the **reboot** sequence command on Linux requires more thought than the older VME systems. Our approach is to provide default functionality that works like the older systems, but to also provide a few other useful options.



After experience with GPI, we are currently reviewing this mechanism. One option under consideration is simply remove the need of a reboot sequence command for Linux-based systems, and use standard init.d scripts.

Argument	Definition
reboot	When the **reboot** sequence command is received by the GMP with the *reboot* argument, the GMP will execute the start/stop/reboot script for the instrument software with the *reboot* option followed by execution of the OS shutdown command with the reboot option. With this option, no instrument software is started.
GMP/gmp	When the **reboot** sequence command is received by the GMP with the *GMP*/*gmp* argument, the GMP will execute the **reboot** with *reboot* argument functionality but it will start the GMP process as part of the reboot. But with this option, only the GMP software will be started.
none	When the **reboot** sequence command is received by the GMP with the *no* argument, the GMP will execute the **reboot** with *GMP*/*gmp* argument functionality followed by the restart of the instrument software via the instrument start/stop/reboot script using the *start* option.

Table 8: Reboot sequence command options on Linux

The **reboot** functionality will be handled through an installed init.d script that will understand how to optionally start the GMP. The GMP init.d script will coordinate the start/stop/restart/reboot of the instrument using another init shell script for the instrument. This script is called the *instrument startup script*.

The instrument builder must provide the instrument startup script written as a traditional Linux init script (preferably in bash or sh) that will take the traditional init.d arguments of: start, stop, restart, status, usage, and reboot. The name of the script should be the agreed upon name of the instrument followed by .sh (e.g. gpi.sh). This script acts upon the entire instrument software system, including instrument software on the TLC and any subsystems (REQ-GIAPI-SC-03). The actions associated with the instrument startup script arguments are described in the following table.

Argument	Definition
stop	The *stop* argument indicates that any executing instrument software should be stopped. All processes should end execution. When the instrument startup script is executed with the *stop* argument the following should happen: 1. The script should stop any and all instrument subsystems. 2. The script should stop the instrument software running on the TLC. The instrument startup script should not exit until all instrument software has been stopped. If the instrument software is already stopped, the script should return immediately with a message and no error.
start	The *start* argument indicates that the instrument software should start all of the instrument software. When the instrument startup script is executed with the *start* argument the following should happen: 1. The script should start instrument software on any and all subsystems. 2. The script should start the instrument software on the TLC once the subsystems are running successfully. The instrument startup script should not exit until all instrument software has been started. If the instrument software is already started, the script should return immediately with a message and no error.
restart	The *restart* argument should be implemented as a *stop* followed by a *start*.
status	The *status* argument should print out a summary of the execution status of the instrument software on the TLC and the subsystems.

usage	The *usage* argument should print a message indicating what each of the options does.
reboot	The *reboot* argument is not a typical init.d script argument. The *reboot* argument causes the stop (if needed) and reboot of the subsystems. When the instrument startup script is executed with the *reboot* argument the following should happen: 1. The script should execute the *stop* option for the instrument software on any and all subsystems if the software should be stopped before a reboot. 2. The script should start the shutdown and restart of each of the subsystems in a way that is appropriate for the subsystem. 3. The script should stop any instrument builder software on the TLC. The instrument startup script should not exit until all steps have been completed.

Table 9: Definition of instrument startup script arguments on Linux

10.3.3 Observation Preparation Sequence Commands

These sequence commands notify the instrument that specific activities are occurring in the overall process of preparing to acquire data. Although the sequence commands are informational, some instruments may need to take actions when they are received. Table 10 lists these sequence commands.

Sequence Command	Arguments	Action/Definition
verify		This command indicates to the instrument that verification of configurations is underway by the OCS, SSAs, and observers. The instrument must be capable of executing changes to its configuration during verify mode (REQ-GIAPI-SC-04).
endVerify		This command indicates to the instrument that verification of configurations is completed.
guide		This command indicates that the instrument should do whatever is necessary to start guiding operations.
endGuide		This command indicates to the instrument that the guiding operations should end.

Table 10: Instrument setup/preparation phase Sequence Commands

The example in Figure 13 shows the use of **verify**/**endVerify** to bracket the setup portion of a science observation. **Verify** occurs just after the time when the target is acquired and ends after the telescope and instrument are ready for the initial science **observe** (i.e. start of step 0). For instance, during this preparation phase, the exposure time may need to be iteratively determined by a acquiring a series of datasets or a different filter may need to be inserted to peak up on a target. The datasets are usually not permanently saved during the verify phase.

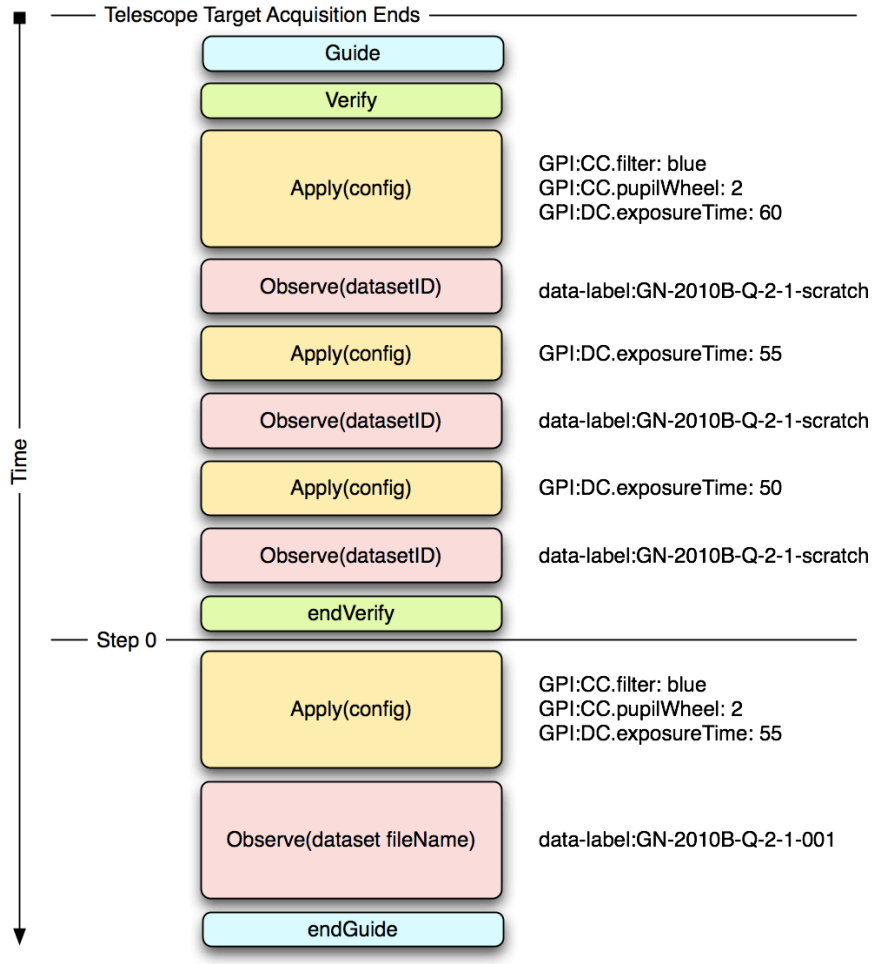


Figure 13: Observation showing use of setup sequence commands

Guide/endGuide are similar, but usually bracket the entire verify phase and science observation. They alert the instrument that the telescope is guiding with whatever mode was configured for the sequence.

The current SeqExec does not send these commands to the instrument, but future versions may so they need to be implemented. An instrument may not require that anything be done for these sequence commands.

10.3.4 Apply Sequence Command

The **Apply** sequence command is the one that contains much of the implementation work for the builder. **Apply** is defined in Table 11 and examples of its use appear in Figure 13.

10

Sequence Command	Arguments	Action/Definition
apply	configuration	The apply sequence command causes the instrument software to match the configuration that has been set to it by the OCS. The configuration is the argument of the **apply**.

Table 11: Definition of Apply sequence command

Apply takes a configuration as its only argument. The configuration contains instrument parameters that are merged with the current instrument configuration to define a new configuration. The **apply** is completed when the instrument successfully matches the configuration or produces an error trying. Most of the remainder of this section discusses how **apply** is implemented by the builder using the GIAPI.

10.3.5 Data Acquisition Sequence Commands

The commands in this section are used to start, stop and control the data acquisition during an observation. Table 12 describes the six data acquisition sequence commands.

Sequence Command	Arguments	Action/Definition
observe	data-label	This command indicates that data acquisition should begin in the instrument based upon its current configuration using the given data-label as the name of the file that should be written. See section 13.1 for information on data-label.
endObserve		This command indicates to all principal systems that the instrument has completed the configured observation.
pause		This command indicates that a system should do whatever is appropriate for it to pause data acquisition. **Pause** indicates to the instrument that the user intends on continuing at a later time. Higher-level software uses the **pause** command to synchronize nodding when an observation requires nodding.
continue		The command is the reverse of **pause**. Higher-level software uses the **continue** command to indicate to the instrument that a nod operation has completed and that it should continue the observation.
stop		This command indicates that a system should stop the current data acquisition process normally at the soonest appropriate time such that datasets are scientifically useful.
abort		This command indicates the instrument should stop the current data acquisition process immediately and discard any data already collected.

Table 12: Data acquisition Sequence Commands

Observe starts the data acquisition process with the instrument in the configuration at the time it receives the **observe**. The argument should be used to create the file for the dataset in the form data-label.fits. The argument may be of some other form than a data-label so the instrument should treat the argument opaquely. (For instance, for testing the argument could be *lab-test-02202007*, which isn't strictly a Gemini data-label.) The GIAPI activities during **observe** are covered in Section 12.3 on page 46.

The following points are relevant to these sequence commands.

- All the sequence commands in this section behave according to the ACM. For instance, the **observe** returns from the acceptance phase when the instrument starts acquiring data. The action of data acquisition is busy for the entire time that the instrument is acquiring data. Completion of the observe action is required following data acquisition.
- **endObserve** is informational at this time. No systems at Gemini take any actions on endObserve.
- Some instruments cannot **pause** and **continue**. For instance, IR detectors cannot pause data acquisition, but an optical CCD can by closing the shutter. If the instrument cannot pause, **pause** should be implemented to throw an error and **continue** should successfully complete immediately (REQ-GIAPI-SC-05). An instrument should support **pause** if possible (REQ-GIAPI-SC-06). An instrument's behavior for pause and continue should be documented in its public interface (REQ-GIAPI-SC-07).
- **Stop** and **abort** are not the same. **Stop** is a request for successful early termination of the observation and **abort** is a request for early termination for a reason that is unknown to the instrument. **Stop** saves useful science data and **abort** should not generate data.

10.3.6 Engineering Commands

The **Engineering** sequence command is a new addition. Its goal is to enable high–level engineering tools (GUIs or command line applications) to send engineering commands remotely to the instrument. **Engineering** is defined in Table 13: Definition of Engineering sequence command.

Sequence Command	Arguments	Action/Definition
engineering	COMMAND_NAME	The engineering sequence command has one mandatory argument, which is the name of the engineering command to be executed. The commands themselves are instrument specific.
	…	Any number of optional arguments can be sent, which will be dealt by the instrument according to its own definition of its valid engineering commands.

Table 13: Definition of Engineering sequence command

10.3.7 General Notes on Sequence Commands

The following are some general comments on sequence commands and related builder requirements.

- Sequence commands are high-level commands that must be implemented in the appropriate way for each instrument. The theory is that the builder knows best how to use the definition of each sequence command and properly interpret it in the context of the operational modes of the instrument. The description of the implementation of each sequence command is part of the description of the instrument's public interface (REQ-GIAPI-SC-08).
- All instruments must be capable of receiving and responding to all the sequence commands (REQ-GIAPI-SC-09). Some of the commands may not be relevant for the instrument. In that case, it must acknowledge the command and indicate immediate successful completion.
- The most important sequence command is **apply**. An instrument receives a demanded configuration as an argument to apply and the instrument software must match this configuration. Implementing **apply** is the most complex things the instrument does and it requires good design and testing by the builder and Gemini.
- Note that an observation sequence consists of **apply** followed by **observe**. This pattern is repeated over and over during observing. Figure 20 shows an example of this.

10.4 Sequence Command Concurrency

Section 10.3 describes the 16 sequence commands that make up the instrument command set. The following are comments about what a builder can expect regarding ordering and concurrency of sequence commands.

- At no time should the instrument enter a state where it cannot or does not accept a command (REQ-GIAPI-SC-10). Accepting means processing a command request.
- The instrument should allow **init** to be executed at any time resulting in the end of any ongoing sequence commands and the beginning of the **init** (REQ-GIAPI-SC-11). This is needed to clear any problems that might occur in execution of actions related to other sequence commands.
- If executing actions from a sequence command other than **init** or **apply**, the instrument should refuse to start the associated actions (REQ-GIAPI-SC-12).
- An instrument must support the concurrent execution of multiple **apply** commands (REQ-GIAPI-SC-13). The instrument must handle the case where **apply** is to a device or subsystem that is already being demanded to a new configuration (REQ-GIAPI-SC-14)
- The instrument should assume that only one **observe** will execute at a given time, and an error should be returned if an **observe** is received while executing **observe** (REQ-GIAPI-SC-15).

10.5 Handling Gemini Sequence Commands in GIAPI

The GMP provides much of the functionality needed to support the reception of sequence commands. 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. This is implemented in a way that is similar to registering interest in a status item in Section 9.5.4. Then, through other GIAPI methods, the instrument provides command completion information to GMP and Gemini.

10.6 Sequence Command Handlers

A Sequence Command Handler (SCH) is an instance of a class that implements a specific interface with one method named `handle`. One SCH can be used for multiple sequence commands or each sequence command can have its own. The signature of an SCH looks like this:

```
HandlerResponse handle(ActionID id, Request request, Configuration config);
```

ActionID – is a unique value that identifies the request or equivalently identifies the set of actions started by the request. The receiver of the command must retain the value for the ActionID unless the actions started by the request are complete immediately (see below). An `ActionID` will not be used again in any future requests until the GMP is restarted.

Request – is an object that contains two enumeration values. The first indicates the sequence command that should be handled. The second indicates the activity requested by the sender. The enumerated values for the sequence commands are listed in Table 14. The enumerated values for the activity are listed and described in Table 15 (and discussed in more detail below.)

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 14: Sequence Command enumeration types

Configuration – The configuration was defined in Section 10.1. **Apply** and **observe** sequence commands contain a non-empty configuration of parameters and values.

The requested activity is listed and defined in Table 15. An activity is always shown in bold italic as in ***preset***. The requested activity requires some explanation. Preset and preset/start indicate what the receiver should do with the request. Preset is the act of evaluating whether or not the requested actions can be started.

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 15: Activity names, enumerated values, and descriptions

Some notes on the requested activity and its use.

- A system must be able to execute a ***preset*** for any sequence command at any time (REQ-GIAPI-SC-16). This suggests that the code that implements the handle method must evaluate requests and dispatch actions. It cannot execute actions directly/synchronously and become unavailable while busy.
- ***Preset*** generally checks to see if there is some reason a sequence command can't start. The check could be based on already executing actions or an evaluation of the parameters in the configuration.
- It is possible that a sequence command will be received when actions related to that command are already executing. The acceptance phase or ***preset*** activity must still execute (REQ-GIAPI-SC-17).

A `HandlerResponse` is returned whenever the `handle` method is invoked, and it must be returned within 300ms (REQ-GIAPI-SC-19). Failure to do so will result in the OCS considering the system unresponsive. `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 (see section 10.2).

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 should be used for all Long commands, defined as those where the time to process them could take longer than 300ms, and therefore the system must return `STARTED` and handle the command completion asynchronously. This is 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 will be processed in 300 ms or less, such as those 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 16: `HandlerResponse` choices and descriptions

The following are notes regarding the `HandlerResponse` and its use.

- The only response that indicates actions have started is `STARTED`. The other three do not result in any ongoing actions that must complete at a later time.
- Some **apply** configurations can be matched quickly without moving devices. An example might be the detector exposure time. A handler is allowed to execute very short actions and return `COMPLETED` immediately with the ***start*** or ***preset/start*** activity.

- All three Request activities can result in an `ERROR HandlerResponse` with an associated cause for the error. For instance, if the ***preset*** activity fails due to a bad argument, `ERROR` should be returned by the `HandlerResponse`.

10.7 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**. **Apply** is unique because it takes a potentially complex configuration. **Observe** includes a single parameter in its configuration but its use is similar to the commands in this group.

Builders must register an implementation of `SequenceCommandHandler` for each of the sequence commands (REQ-GIAPI-SC-18). The following shows the GIAPI method to be used for sequence commands other than **apply**:

```
void subscribeSequenceCommand(SequenceCommand id, Activity[] activities,
                              SequenceCommandHandler handler);
```

- The `SequenceCommand` is the name of the type for the enum values of Table 14.
- The second argument is an array of `Activity` enum values from Table 15. This is to support using different handlers for different activities.
- The most recently registered SCH for a sequence command will always be used.

Providing action completion information for these sequence commands is covered below in section 10.9.

10.8 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. For instance, the configuration could have parts for the IS and DC, but many choices are possible based on the instrument implementation. GIAPI allows many handlers to be associated with **apply**; each associated with a part of a configuration. Naming in a configuration is discussed in the next section.

10.8.1 Configurations and Parameter Naming

The **apply** sequence command has a single argument that is a configuration. A configuration is a set of parameters and values that are treated as a unit and passed to the instrument at once. Parameters must have unique names within the Gemini control system following the GIAPI naming conventions of section 8.1.

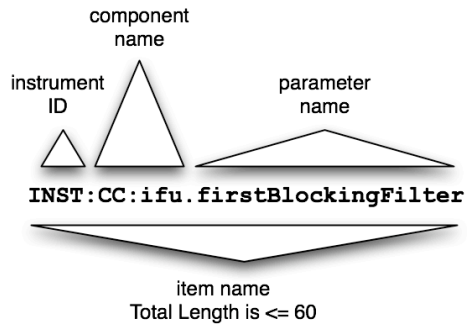


Figure 14: Command parameter naming

An example configuration for the **apply** is shown in Figure 15. Parameters are present for the CC and DC. A parameter *prefix* is defined to be the instrument ID + component name + any other hierarchy that is common to a set of parameter values.

```
NIFS:CC.firstBlockingFilter = blue
NIFS:CC.mask = 1
NIFS:CC.grating = GEM4323
NIFS:DC.exposureTime = 22
NIFS:DC.exposureTime.units = seconds
NIFS:DC.bias = 0.234
NIFS:DC.bias.units = mv
```

Figure 15: Example apply configuration with two parts

In Figure 15 the three possible prefixes are NIFS, NIFS:CC, and NIFS:DC. The prefix is used in the next section.

10.8.2 Subscribing to Apply in GIAPI

The registration of a handler for **apply** uses a specific GIAPI method that assigns a SCH to a configuration prefix and `Activity`.

```
void subscribeApply(String prefix, Activity[] activities,
                    SequenceCommandHandler handler);
```

When the GMP receives a configuration it will split up the configuration according to the registered handlers and pass each the part of the configuration that it is registered to handle.

The `Activity` array allows different handlers for different activities. For instance, one handler can implement only ***preset*** and another ***start***. In this case, the GMP ensures that that the two activities are executed together as an indivisible unit. Separate handling of **preset** and **start** allows a builder to be flexible as is shown in an example in Section 10.11.

The following Java code shows how to split the handler of configurations between a handler for DC and a handler for CC for the example in Figure 15.

```
void subscribeApply("NICI:CC",
                    new Activity[]{Activity.PRESET, Activity.START}, handler1);
void subscribeApply("NICI:DC",
                    new Activity[]{Activity.PRESET, Activity.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.

Since handler1 and handler2 are registering for both ***preset*** and ***start***, it makes more sense to register for ***preset/start***. This reduces the number of calls to the handler. This situation is the reason to use ***preset/start***.

```
void subscribeApply("NICI:CC", new Activity[]{Activity.PRESET_START}, handler1);
```

Preset/start is an optimization that indicates to the GMP that the builder will handle ***preset*** and ***start*** in the same SCH. From the GMP point of view, if a handler is registered for the ***preset/start*** activity, GMP will make one call to the handler rather than two. Similarly, if the same handler is registered for `Activity.PRESET` and `Activity.START`, the GMP will call the handler with `Activity.PRESET_START` when possible.

Stating this in a slightly different way: subscribing to ***preset/start*** does not mean that the handler will always and only be called with the ***preset/start*** activity; it is a short-cut for subscribing to ***preset*** and ***start*** with the same handler. There are times when the GMP must call a handler with ***preset*** and then ***start*** separately even when it is registered for ***preset/start***. This occurs when the GMP must execute a configuration with some handlers that are ***preset/start*** and some that have separate ***preset*** and ***start***.

10.9 Updating Gemini with Action Status

The last idea 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 has the following signature.

```
void postCompletionInfo(ActionID id, HandlerResponse response);
```

The arguments are all described in previous paragraphs.

ActionID – the completion information is associated with a set of actions identified by an `ActionID`, which is passed to the instrument through the handler.

HandlerResponse – when used in `postCompletionInfo`, the response should be `COMPLETED` or `ERROR`. If the actions end in error, a message should be included in the `HandlerResponse` indicating the reason for the error.

Some usage notes:

- Although it can be, it's not necessary that the SCH be the piece of software that also sends the completion notification message to GMP. Splitting the responsibilities allows a scenario where the code that starts the actions resides on the TLC, but the code that executes the action and notifies completion is on a different thread or even computer.
- With multiple handlers handling different prefixes, different handlers must contribute completion information for the part of a configuration for which they are responsible. The GMP receives and merges completion information for an `ActionID` from the registered handlers, notifying the OCS when the entire configuration is completed successfully or with an error.
- If all the actions are started, the handler only needs to keep the most recent `ActionID` and track all its ongoing actions–this includes actions from all previous requests as well as the current one. Only when all the handler's ongoing actions are completed does the handler report `COMPLETED` using `postCompletionInfo` and the most recent `ActionID`. Completion of the most recent actionID means that actions with all other previous actionIDs are also completed. When the handler posts `COMPLETED`, all actions the handler is responsible for are idle. This greatly simplifies the program logic. This approach is exactly what is done in the EPICS system, and it has worked adequately.

10.10 Dealing With Apply Errors

This section describes how errors should be handled with the **apply** sequence command in the ACM and GIAPI.

10.10.1 Acceptance Errors

Each SCH has responsibility for matching configurations with a specific prefix (see Section 10.8). There are two phases to responding to an **apply** sequence command (see Section 10.2). The first phase, called acceptance, is started with the ***preset*** activity. During ***preset*** the handler determines if there is a problem with the configuration. If any part of the configuration can't be executed the handler should return `ERROR` with a message using the `HandlerResponse` (see Section 10.6).

Like ***preset***, the ***start*** should fail if all actions can't be started, the handler should return `ERROR` with a message using the `HandlerResponse` (see Section 10.6).

Error considerations during acceptance when using the ***preset/start*** activity are the same as separate ***preset*** and ***start***.

10.10.2 Immediate Completion

In some cases all the actions can complete successfully very quickly. The handler indicates the actions associated with the `ActionID` are completed to GMP. These types of commands generally don't fail. However, the statements of the previous two sections are also valid for this case.

10.10.3 Ongoing Actions

When the handler receives a second request while actions are already ongoing, it must evaluate the ***preset*** and ***start*** in light of the ongoing actions. It must be possible to handle ***preset*** at any time (see Section 10.6), but it may or may not be able to ***start*** all the actions. If the handler can't start all the actions, the handler returns an ERROR to GMP through the `HandlerResponse`. The entire configuration must be rejected if one piece can't be started. If all the actions are started, the handler returns STARTED. The acceptance phase is then completed.

Once the actions are started the GMP is made aware of the completion status of the actions through the `postCompletionInfo` call.

10.11 Suggested Usage Patterns

This section describes good ideas for using the sequence command functionality in GIAPI.

10.11.1 Single Preset Handler

It is possible to have several sequence command handlers (possibly on different machines) so reducing the number of messages that must occur to start a command's actions is a good goal. In the worst case, a configuration with N parts, each part processed by its own handler, requires 2N messages to start. This approach is what is done in EPICS where each CAD record supports a small part of a possible configuration, and the apply record does a preset and start on each CAD. This will work but there are ways to do things differently that are more efficient.

The best possible case with N messages is obtained by using N handlers and the ***preset/start*** activity. The GMP will always try to use ***preset/start*** if handlers for the activity are registered. This optimization can also be used if there is only one part in the configuration.

With both ***preset*** and ***start*** and N handlers, a two phase approach requiring 2N messages is needed. The first N messages are used to evaluate ***preset***; the second N messages evaluate ***start***. ***Preset*** must evaluate the entire configuration before starting any of the actions in the second phase.

Figure 16 shows a usage pattern that can use the GIAPI features to limit the number of messages for N command handlers to N+1 messages.

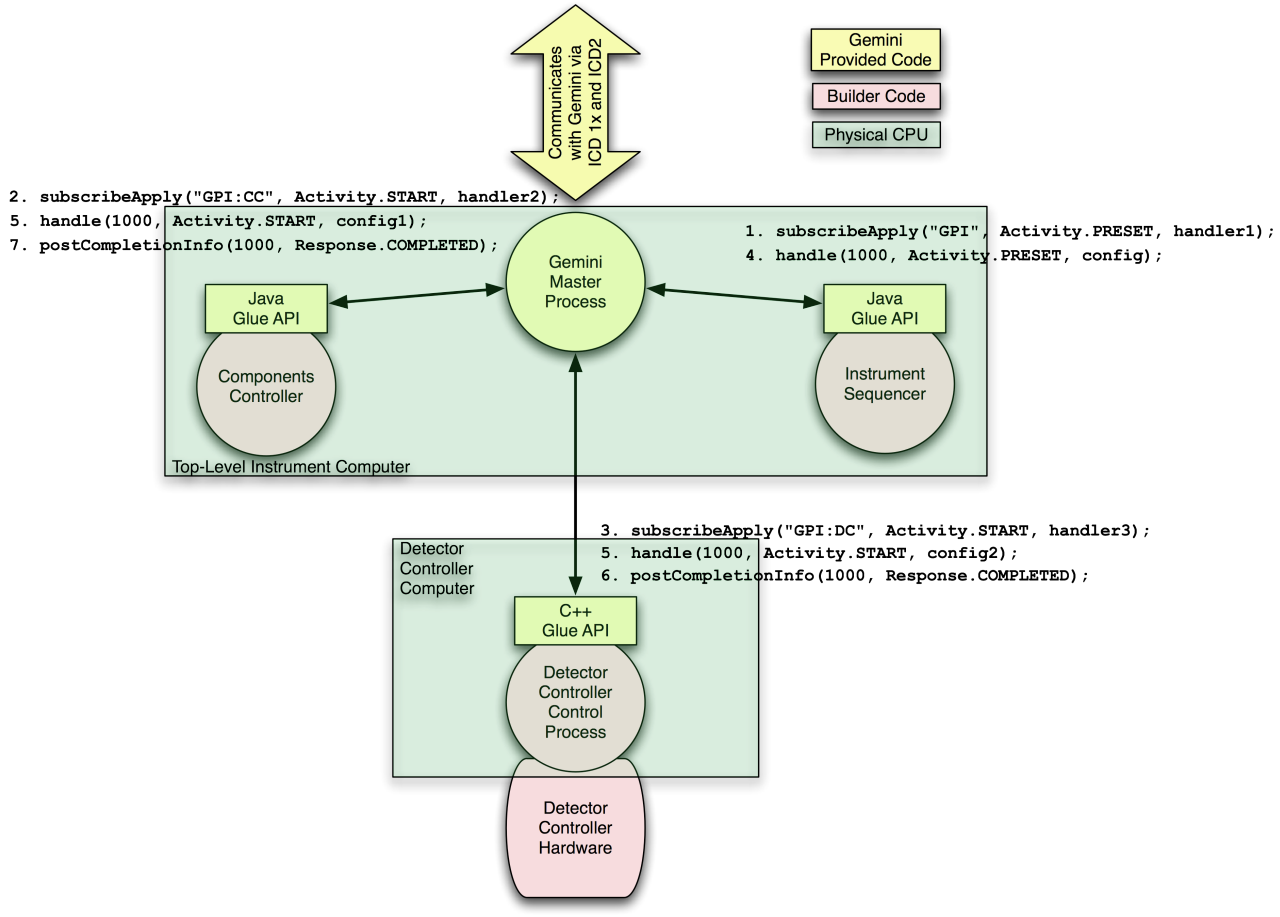


Figure 16: Best practice distribution of command handling responsibilities

The figure shows an instrument deployed across two computers. The IS and CC run on the TLC along with the GMP. The detector controller hardware and software are deployed on a dedicated second computer.

The numbers in the figure show the order of GIAPI calls for this example. The three components subscribe to specific activities in 1, 2, and 3. The IS subscribes to *all* ***preset*** activities in 1. The CC and DC subscribe to the ***start*** activity for prefixes NIFS:CC and NIFS:DC in steps 2 and 3, respectively. Then the configuration to the right is received from the OCS by the GMP.

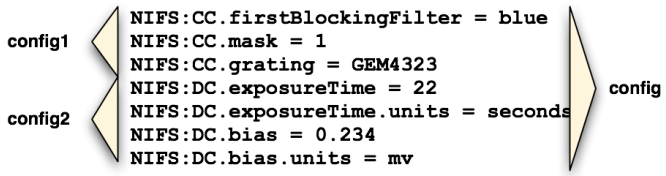


Figure 17: Demonstration configuration

In step 4, the entire configuration is passed in a message to the IS for the ***preset*** activity. The IS has access to the current configuration of the entire instrument and is uniquely able to determine if the instrument can go ahead and match the configuration. It returns `ACCEPTED` to the GMP indicating that the actions can be started.

In steps 5 and 6, the GMP sends a message to DC and CC to start the actions. The entire configuration has now been split into the parts each handler has agreed to process. This is shown as config1 and config2 in the small figure above. The DC and CC start the actions and return `STARTED` to GMP. (In this case, the DC could probably have set the exposure time and bias voltage immediately and returned `COMPLETED`.)

The code in the DC and CC executes the actions. The processes in the DC and CC complete the actions and send completion information to the GMP in steps 6 and 7. The total number of messages is 1 ***preset*** plus N ***starts***.

The GMP receives the completion information and merges it based upon the `ActionID`. Once it has received all completion information it updates the sender in the OCS.

10.12 Private Commands E

The use of JMS enabled private status in Section 9.5.4. The use of JMS can also enable a private command infrastructure within the instrument. The infrastructure described in this section works properly across multiple machines and therefore supports the ability to send private **apply** sequence commands between computers.

Two methods will be added to the GIAPI to support posting sequence commands internally; one called `postInternalApply` for the **apply** sequence command and `postInternalSequenceCommand` for the other sequence commands.

This feature should only be used as part of an agreement to use GIAPI as the internal infrastructure for the instrument.

11 Interacting with Gemini Systems

Some instruments need to interact with other Gemini principal systems. Examples are listening to the current altitude of the telescope, receiving tracking information from the Acquisition and Guide Unit for an instrument wavefront sensor, sending an offset to the telescope, or offloading corrections to the PCS through the TCS. EPICS is the backbone of the Gemini run-time telescope environment. This section describes the parts of the GIAPI that allow an instrument to do these kinds of things.

GIAPI does not provide open access to the TCS or other systems. Features are added as needed so that we understand and control the interface between instruments and principal systems.

11.1 Receiving Status Information From a Gemini System

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

The model for providing this capability is based on the GIAPI ideas presented in Section 9 on status. The GIAPI allows the builder to subscribe to a status item with the name of a current TCS channel. Figure 18 shows how the OSGi-based GMP provides support for this feature.

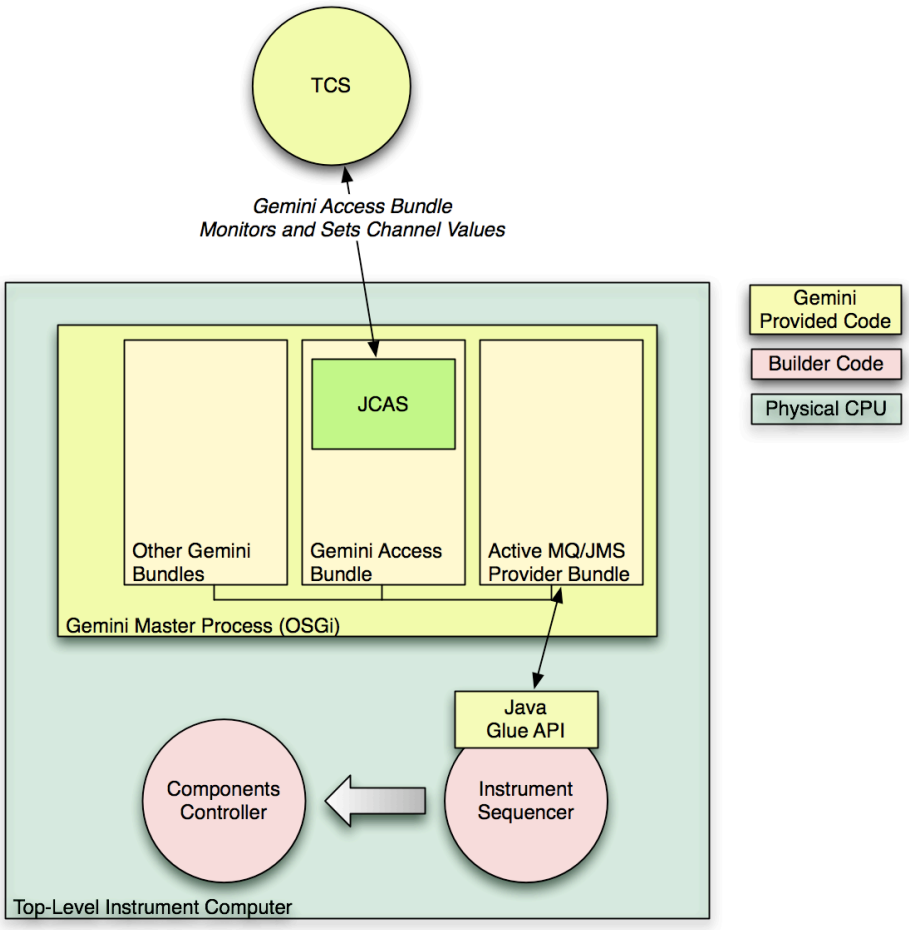


Figure 18: GMP Showing Gemini Access Bundle

The Gemini Access Bundle, a bundle available in the GMP process, provides subscription and updating from the TCS as necessary. The Gemini Access Bundle is based on Java Channel Access. Instruments that don't need this functionality don't include the bundle.

11.1.1 GIAPI EPICS Status Monitoring Functionality

Here are the methods used to access EPICS status. The caller registers and update method for a specific EPICS item. The update method is called when the EPICS system publishes an update.

`subscribeEPICSStatus` – This method is called with the name of a status item to receive updates. It takes as arguments the item name and a method that will be called when the item is updated. The argument of that method is the StatusItem object.

`unsubscribeEPICSStatus` – This method is called with the name of a status item to receive updates. It takes as arguments the item name.

The names of status items that are possible is agreed upon during the instrument design as part of the Instrument/Gemini interface (REQ-GIAPI-GI-01).

Common TCS information related to the current position of the telescope is available in the TCS context (See Section 11.4.) Using the method in Section 11.4 is more appropriate when a calculation must be performed occasionally (rather than frequently) or the calculation needs much of the information in the TCS context (such as an instrument flexure correction.)

11.1.2 Monitoring Scenario

An instrument with an active flexure model needs to be kept up to date with the value of the telescope altitude. This is done by calling subscribeEPICSStatus("tcs:sad:currentEl", MyUpdateMethod). The method MyUpdateMethod will be called as the current altitude changes.

11.2 Instrument OIWFS Updates from AGCS

Instruments with an On-Instrument Wavefront Sensor (OIWFS) must receive tracking demand data from the TCS at 20 hz. The tracking data consists of position values with timestamps for the time the information was sent and the time when it should be used. The tracking data arrives from the TCS via EPICS Channel Access.

The model for providing this capability is being researched and will be refined as more details on new instruments become available. The following are probably true:

- The wavefront sensor subsystem may or may not be part of the instrument contract. Control for this device will often be based in a computer that is separate from the instrument TLC.
- It hasn't been tested, but it doesn't make sense to push 30hz updates through the GMP infrastructure for this specific application; it just adds complexity and reduces testability.
- The use of the EPICS bundle in the GMP should work, but it would put a lot of software infrastructure into a situation that has a fairly high performance (bandwidth) requirement and more importantly a requirement for predictable latencies.

Therefore, the solution will be to build a small library of C or C++ code that will appear as an EPICS system to the TCS that implements the EPICS interface. The library will have a simple C-level interface for the builder to receive updates. This Channel Access Server will reside on the computer responsible for the control of the OIWFS position. The package will provide the functionality needed to integrate with the TCS and should allow modular testing of the subsystem.

11.3 Making Information Available to the TCS

Instruments occasionally need to interact with the TCS by delivering information to the TCS or one of its sub systems. At Gemini, the TCS is the public interface for all subsystems and it provides interfaces to features as needed by other systems. This section accumulates the known instrument to TCS needs.

11.3.1 Offloading To the Primary Mirror

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 TCS can accept this kind of update through the AGCS. The instrument must post an array of Zernikes which will then be captured by the AGCS, merged in the TCS and passed to the PCS.

The GIAPI provides a method the instrument can use to update the PCS through the TCS. Units are generally rms wavefront errors in microns, but these details will be specified in the language ICDs.

`postPCSUpdate` – This method is called to offload corrections to the PCS. It takes as arguments an array of Zernike coefficients in correct units.

11.4 World Coordinate System Interactions

The science dataset must be calibrated to allow conversion of focal plane x, y positions to right ascension and declination. The conversion is dependent on the specifics of the instrument and telescope configuration at the time of the dataset's acquisition (although many of the parts of the conversion are constant.)

Currently, this 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 [10]. 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.

Providing WCS information for dataset headers is an instrument requirement (REQ-GIAPI-GI-02). The GIAPI provides a method that can be called to fetch the TCS context in all language glue APIs. There is no plan to recode the C libraries into any other language so complex transformation will need to be done in C (this may change in the future).

`getTCSContext` – This method is called and returns a TCS context object with values that are up to date at the time of the call.

Instrument code should make the `getTCSContext` call during the `OBS_PREP` phase of the **Observe** prior to starting data collection (See Section 13.4. There are two possibilities for handling the results of this calculation: publish the WCS coefficients as status items, or place the WCS information directly into headers. The decision on this is TBD; both require essentially the same instrument software.

Table 17 contains the elements present in the TCS context as available directly from the TCS. The `getTCSContext` may provide a subset of the most useful information depending on builder need.

Context Index	Context Item Name	Description
0	ctx->time	Time stamp (TAI)
1-3	x, y, z	Cartesian elements of mount pre-flexure az/el
4	ctx->tel.fl	Telescope focal length (mm)
5	ctx->tel.rma	Rotator mechanical angle (rads)
6 7	ctx->tel.an ctx->tel.aw	Azimuth axis tilt NS (rads) Azimuth axis tilt EW (rads)
8	ctx->tel.pnpae	Az/el non-perp (rads)
9 10	ctx->tel.ca ctx->tel.ce	Left-right collimation (rads) Up-down collimation (rads)
11-24	ctx->aoprms	Target independent apparent-to-observed parameters
25-30	ctx->m2xy	M2 tip/tilts
31 32	ctx->po.mx ctx->po.my	Mount pointing origin in X Mount pointing origin in Y
33 34 35 36 37 38	ctx->po.ax ctx->po.ay ctx->po.bx ctx->po.by ctx->po.cx ctx->po.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
39-44	ao2t	Optical distortion coefficients (not used to date)

Table 17: Contents of the TCS Context as received directly from the TCS

`getTCSContext` can also be used when the instrument needs to make occasional calculations unrelated to WCS that require much of the information in the TCS context. The advantage of this approach is that the context contains current information from the telescope at the time of the call.

11.5 Controlling the Gemini Secondary

Many instruments with an AO component need to directly control the Gemini secondary. This interaction is typically very fast and requires a high-performance, low-latency connection to the Secondary Control System (SCS). The hardware-based system providing SCS access at Gemini is called the Synchro Bus.

Currently there are two ICDs related to the Synchro Bus: ICD 10 and ICD 20. ICD 10 describes the EPICS driver interface to the VMI VME-based reflective memory board. ICD 20 assigns node numbers and reflective memory pages to systems accessing the bus.

In the GIAPI, instruments accessing the Synchro Bus will continue to use a C-based library. The library interface for Linux is documented in ICD 10. The following are some points regarding Synchro Bus use:

- GIAPI will provide Linux-based instruments with a C-level library/driver to access the Synchro Bus. ICD 10 has been updated to reflect the addition of Linux access.
- All instruments are required to use the Gemini-provided Syncro-bus library (REQ-GIAPI-GI-03).

- The library is a standalone component of the GIAPI. It will be linked with builder code on the system that accesses the Synchro Bus. No communication with the GMP is needed.
- At a future date, Gemini will specify a reflective memory board that is compatible with the current reflective memory bus and the GIAPI TLC hardware specifications. This hardware choice will be made when the first instrument and Gemini instrument managers decide that it is necessary.
- ICD 20 must be modified to assign reflective memory pages to each new instrument.
- NOTE that synchro bus communicates using the Motorola endian ordering. Note this is primarily a problem for Gemini.

12 Using Gemini Services

Gemini Services are features of the Gemini environment that most instruments will need. When an instrument needs a service like one provided by GIAPI, the GIAPI service should be used (REQ-GIAPI-SVC-01). It is anticipated that new services will be added as GIAPI is used.

12.1 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 based on priorities. System logging is a service supported by the GMP that allows any process to contribute to a system log.

12.1.1 Process Logging

Every program benefits from an ability to provide output that indicates program activities and progress during debugging and sometimes during operations. GIAPI calls this process logging to focus on the fact that it is local to a single process.

The logging operations are usual tied to log levels. Output of logging messages is configurable at runtime. GIAPI will standardize on a logging library for each language API. For instance, for the Java language glue, the built-in Java logging feature will be the standard.

12.1.2 System Logging

System logging allows any process in the instrument to send a logging message through the GIAPI that will then be merged into an instrument-wide log. 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.

`systemLog` – At this time there is expected to be a single systemLog method that takes a log level and a string message. The planned logging levels are ALL, INFO, WARNING, and SEVERE. For example:

```
systemLog(Level.INFO, "WCS information was incomplete.  Missing CTYPE1")
```

The GMP configuration files allow output configuration of the system log file. Processes that contribute to the system log will need to be configured for system logging individually.

The builder is not required to use the process logging features, but the GIAPI will use it and it is available and the builder may prefer to use it to integrate instrument logging and GIAPI logging on the TLC.

12.2 Access to Observatory Time

Access to the current time that is standardized and synchronized across the observatory is essential. Each Gemini site provides a GPS-based time distribution system based upon IRIG-B between a master and system slaves that provides these features. The specified accuracy of the current system is +/- 10 microseconds. All current TCS systems and instruments use this system. The current hardware system and EPICS support is described in ICD 9 [12].

For GIAPI there will be two supported approaches. The first is a software-only system; the second is based upon the current hardware implementation. The expectation is that instruments will only need the software-only system.

Working with the GPI instrument team, it has been decided that the software-only approach will be implemented and used until the hardware-based approach is necessary.

12.2.1 Software-Only System

Gemini is currently in the process of purchasing an integrated GPS/NTP server for each site. This system provides a Stratum-1 NTP server with suggested accuracy to UTC via Ethernet of +/- 50 microseconds or less (depending upon vendor). While this is not as good as the current solution it requires no new hardware or software for proper operation.

12.2.2 Hardware-Based System

If the application requires the hardware-based system's accuracy, a board will be specified and the GMP will be integrated with the board as in the following Figure 19. A bundle will be created as part of the GMP that accesses the board and makes the time available for the instrument.

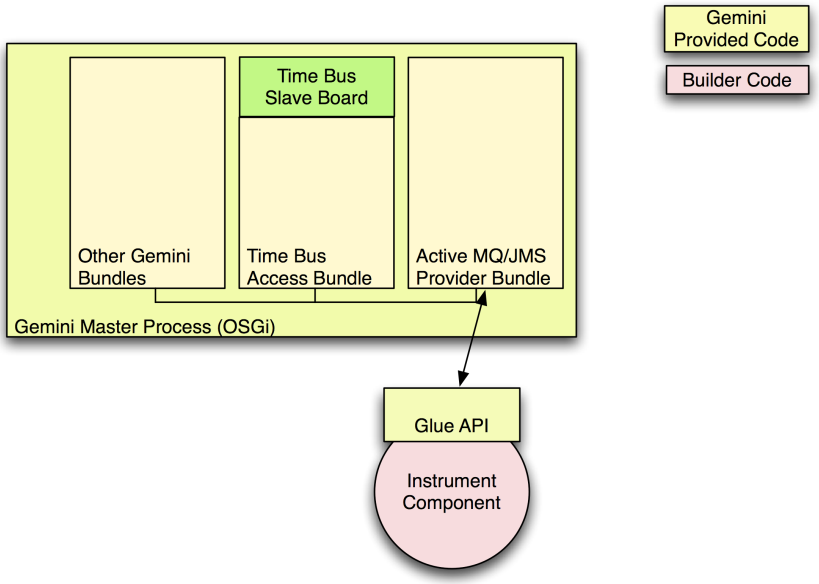


Figure 19: GMP integrated with Time Bus Hardware

12.2.3 GIAPI Time Support

The GIAPI provides the same method to access the time regardless of the internal implementation. The single required method returns the UTC time as the number of milliseconds since January 1, 1970. Some other convenience methods may also be added.

`getObservatoryTime` – This method is called and returns the number of milliseconds between the current time and midnight, January 1, 1970 UTC as a long.

12.3 GIAPI 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.

`getProperty` – This method returns a GMP/GIAPI configuration value based on a String key value.

An example follows:

```
String path = getProperty("GMP_HOSTNAME");
```

In this example, the `getProperty` call returns the host name of the machine running the GMP. The keys for properties are defined in the language ICDs.

13 Providing Science Data to Gemini

The product of most science instruments is science data of some sort. The science data collected in the instrument must be transferred from the instrument to the Gemini software system where it is stored, tracked,

and delivered to observers and the Gemini Science Archive. The Gemini principal system that handles these duties is the Data Handling System (DHS).

The transfer of data from instruments to the DHS has been simplified for GIAPI-based instruments. Gemini has purchased and installed a high-performance Network Attached Storage (NAS) system at both telescope sites that will form the backbone for data transfer between GIAPI-based instruments and Gemini software infrastructure. This system, made by Network Appliances, currently provides 3 terabytes of RAID 5 storage and is scalable to handle future data storage needs. In subsequent sections of this document, this data storage system will be referred to as the Gemini Data Storage Network (GDSN).

The GDSN greatly simplifies the instrument task of storing data at Gemini. Allowing instruments to directly write data to a shared disk reduces the complexity of the data transfer code builders must produce. This is a familiar unobtrusive approach that generally adds no new work for the builder.

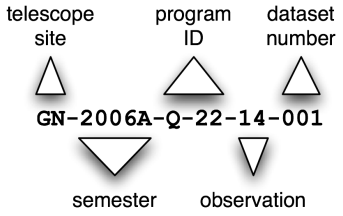
This section describes background for producing science data for instruments using the GIAPI. The following section contains scenarios that show how to use GIAPI.

The current ICD 3 DHS interface is no longer supported and is not an available option for GIAPI-based instruments. The approach presented in this section removes the dependency of the instrument on the DHS software infrastructure.

13.1 The Gemini Data Environment

Each science dataset is written as a single FITS [9] file. Each dataset has a unique *data-label* that looks like: GN-2004A-Q-23-2-001. GN-2004A-Q-23 is the science program ID, the 2 is the observation within the program, and the 001 is the first dataset produced by the observation sequence. Gemini ensures the uniqueness of the data-label.

Figure 20 shows the interactions with the instrument during the execution of a sequence. This figure is used to make some points about the generation of datasets at Gemini. An observation's sequence in the OT is unrolled into a series of steps in the SeqExec program. The example in the figure shows 4 steps. Time is increasing down the page. There are several points based on this figure:



- Each step consists of a configuration phase followed by a data acquisition phase.
- Each step usually consists of the execution of two Sequence Commands (see Sections 10, sequence commands are in **bold**): **apply** followed by an **observe**. But with a constant configuration, the step only includes an **observe** (step 3) (the configuration phase is empty).
- The **apply** requires the instrument to match a configuration. The first step (Step 0) will always provide a complete configuration of the instrument. Complete does not mean it includes every parameter in the instrument. The configuration consists of the items that are presented to the user in the OT.
- The configurations that accompany **apply** sequence commands following Step 0 contain only changes to the configuration set by previous steps. For instance in step 1, the filter and exposureTime are changed, but the pupilWheel continues with value 2. In step 2 only the filter value is changed.
- The **observe** can follow the **apply** as soon as the instrument has successfully completed the **apply**. The **observe** includes a single parameter in its configuration that is the name the instrument should use to name the step's dataset file on the GDSN.
- Each step contains exactly one **observe**.
- One **observe** action creates exactly one dataset and FITS file. Note that it possible for the dataset to have multiple pieces of data as long as the instrument configuration stays the same between readouts and the data is written to a single FITS file.
- The sampling of header information by the OCS wraps the data acquisition part of the **observe** sequence command action (see Section 13.4.2). Another way of looking at the previous point is that to ensure that the sampled configuration data for the header is correct, the configuration cannot change during the execution of an **observe**.

- If there is no change in the configuration between steps, the **apply** sequence command is skipped. The instrument will only receive the **observe**.

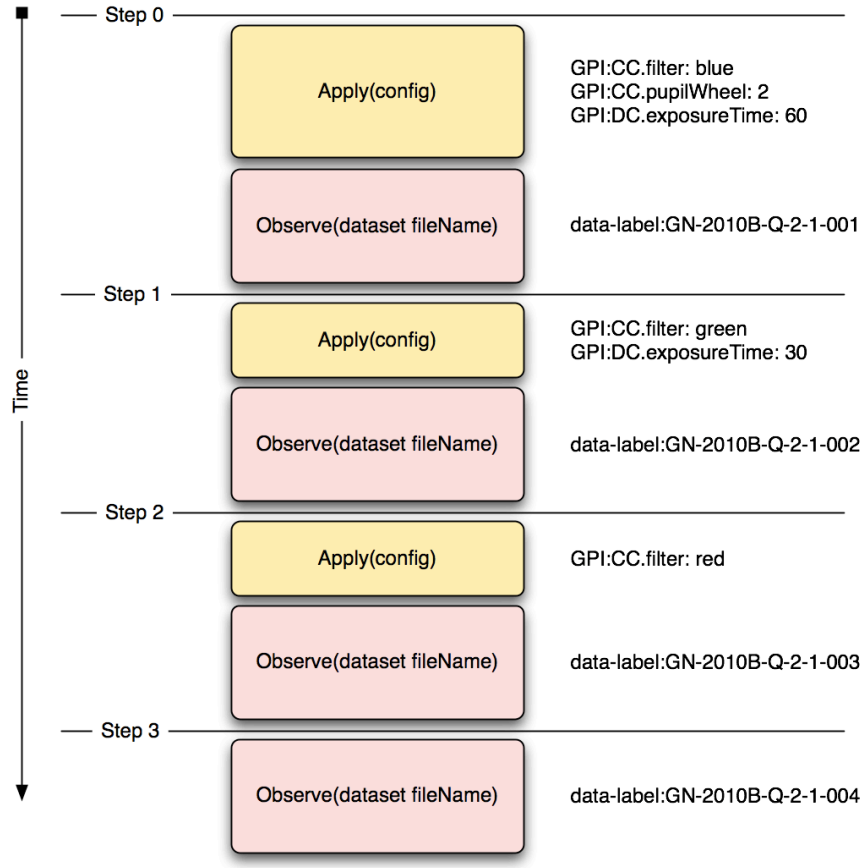


Figure 20: Unrolled sequence showing Observe sequence commands

13.2 Writing Datasets

Instruments must write datasets in two modes: engineering/testing and operations at Gemini (REQ-GIAPI-DP-01). In both modes, instruments write datasets in FITS format using what should be identical code. Having a separate engineering mode allows builders to create headers for testing that are appropriate for their needs.

The instrument writes datasets directly on the GDSN as FITS files with a minimal primary header during operations (see below). The majority of the header information for the datasets is gathered and written by Gemini software (the SeqExec at this time) from the public status items of the instrument and other telescope systems. GIAPI requires that all status items are up to date at all times (see Section 9.5.5), so the set of all status items forms a correct snapshot of the configuration of the entire software system at any given time. The Gemini values are from EPICS systems are always up to date.

The OCS samples header information when two specific events occur: the start of the collection of data and the end of the collection of data. The instrument signals Gemini of these events during the data acquisition process (see below) using the GIAPI.

13.2.1 Datasets as FITS Standard Files

A valid FITS file consists of at least one HDU (Header/Data Unit) that is made up of a Primary Header Unit followed by an optional Data Unit. Each header or data unit is a multiple of 2880 bytes and is padded to the required length as appropriate. The header unit is a sequence of fixed-length 80-character records.

The primary data array is the initial data unit and follows immediately after the first header unit; the header unit is required, but the data unit is optional. A dataset may require additional HDUs following the first, required HDU. These additional HDUs are called conforming extensions. The FITS files written by an instrument must

conform to the FITS standard (REQ-GIAPI-DP-02). More information about the FITS standard is available at [9].

While it is possible to allow builders to develop their own FITS file writing code, it would be a poor use of resources given that well-maintained libraries already exists. To ensure FITS file compatibility, to reduce costs, and to provide a known set of capabilities, instrument builders are required to use the Gemini standard FITS production library specified in the Builder Requirements (REQ-GIAPI-DP-03).

The library chosen for this purpose is the CFITSIO or CCFITS when using C++ code (http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html). This is the "industry standard" package for writing FITS files. Files will need to be specified and written to use the capabilities of this standard library. This is another change that Gemini has made in order to stay out of the development plan of the builder's instrument software.

At this time we have not specified a Java-based FITS writing package. This decision will be made if a builder needs a Java-based capability.

13.2.2 Writing FITS Headers At Gemini

Instrument builder software will write FITS files to a file system that appears as a local file system during engineering at the builder's site and when the instrument is installed at the Gemini site. Datasets should be written at Gemini with a minimal header unit and any minimal extension header units that are needed as required by the FITS standard. The contents of this header are determined during the design process for the instrument (REQ-GIAPI-DP-04).

The complete FITS header for a dataset will be constructed by the DHS from the HDU written by the instrument and the header information collected by the DHS. The DHS is notified of new instrument datasets through the use of Observation Events sent by the instrument using the GIAPI (see Section 13.4).

13.2.3 Writing FITS Headers During Engineering/Testing

During engineering/testing a builder may find that the minimal header is adequate for testing. However, it is reasonable to write a special engineering header that contains additional information suitable and helpful for engineering purposes. If the instrument software chooses to support an engineering mode FITS header in addition to the operations mode FITS header, it must be possible to control this through the ATEUI and instrument configuration (REQ-GIAPI-DP-05).

13.2.4 GDSN Issues

Instruments write their datasets directly to the Gemini Data Storage Network. A storage volume is mounted on one of the instrument computers; either the TLC or another computer in the instrument computing environment (REQ-GIAPI-DP-06). The following are GSDN assumptions, points, and requirements:

- The name for the volume in the GSDN that should be used for writing data will be available as a GIAPI property (see Section 12.3). The specific key values for this property are in the GIAPI programming language glue ICDs.
- Builders can assume there will always be adequate disk space available for the storage of their datasets. Of course builder software allowed to check there is enough disk space before writing your datasets. The point of this bullet is to state we have enough disk space so don't worry.
- Storage for intermediate results, configuration, or other non-data files or temporary files should be kept on storage local to the instrument; the GSDN is used only for science datasets or ancillary files (REQ-GIAPI-DP-07).
- The builder should not depend on processing systems outside of the instrument to apply algorithms to unscramble data from detectors or to assemble data into complete and correct datasets from the data of individual detectors. The details of this kind of processing are to remain within the instrument software. (REQ-GIAPI-DP-08). This is not a new requirement, but it may be an important design issue.
- Any ancillary files that need to be associated with science data and possibly stored in the GSA must follow identical procedures as with science data for defining header content and data unit format. (REQ-GIAPI-DP-09).

13.2.5 Storage for Instruments

The bullet above suggests that instruments should store their temporary files on a disk local to their system and this is the preferred approach. It is also possible to provide disk storage for temporary or non-science results via the network to dedicated disk systems located in the summit computer rooms. Please discuss this with your Gemini representatives.

13.3 Science Data HDU Definition Process

The contents of all FITS files must be a product of the instrument design process. There are two things that are needed:

- Gemini and builders agree upon the instrument status items that must be publicly available for science headers (REQ-GIAPI-DP-10).
- Gemini and builders must agree upon the structure of the data part of the dataset FITS file (REQ-GIAPI-DP-11).

Currently the first bullet is covered by completing a Parameter Definition File (as in ICD 16).

The plan is to update this ICD or create a new one to define a process and format for the results of these decisions.

13.4 GIAPI Support for Writing Datasets and Ancillary Files

The actual writing of a dataset or ancillary file is simple: use the FITS library to write your internal format data as a valid FITS file no the GDSN. GIAPI supports this process by specifying a specific FITS library. However, there is a little more work needed to integrate dataset writing with Gemini, and this section discusses what instruments must do to provide this support.

A science dataset is created and written during the execution of an **observe** Sequence Command. During an **observe** most instruments follow a similar process. The GIAPI requires instrument software notify Gemini that it has reached and started specific phases of the process.

13.4.1 Observation Events

The GIAPI requires information be provided by the instrument to signal important standardized events in the instrument during observation execution. These signals are called Observe Observation Events (OEVT). The GIAPI provides methods for sending OEVT related to the phases of execution during **observe** execution.

13.4.2 Phases and Observe Observation Events in GIAPI

The Figure 21 shows detail of what occurs during the execution of an **observe** on an instrument. Time is increasing down in the figure—the **observe** starts at the top and ends at the bottom. Each phase is indicated by a horizontal line. Table 18 has a complete description of each phase and indicates some of the actions that Gemini takes in response to the delivery of the events.

As with all Sequence Commands, **observe** actions are busy during the entire time of execution. A detailed description of each phase is given in Table 18. The definitions are flexible to allow appropriate interpretation for different instrument/detector types. Events must happen in the order specified in the figure and table (REQ-GIAPI-DP-12).

As each phase is reached, the instrument sends an Observation Event to Gemini with the `postObservationEvent` method. The instrument code must be constructed to allow the events to be fired at the correct times in an execution of the **observe** (REQ-GIAPI-DP-13).

This method takes two arguments. The first is an enumerated type that indicates the specific event. The flags are shown in column 2. The second argument is the data-label that is being generated by the **observe**. For example:

```
postObservationEvent(OBS_START_ACQ, "GN-2010B-Q-2-1-003")
```

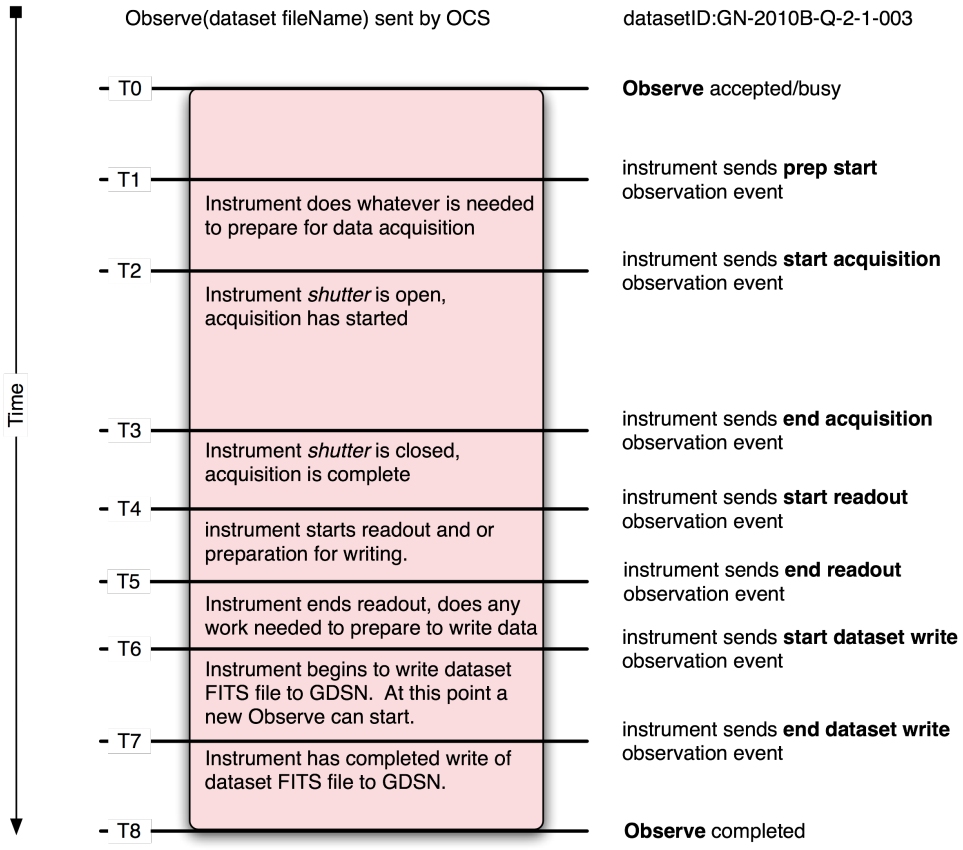


Figure 21: Expanded Observe Sequence Command shows times for observation events

Event Time	GIAPI OEvt	Description	Gemini Response
T1	OBS_PREP	OEvt sent as instrument starts preparation for starting acquisition of a dataset. For instance, this could include pre-reads or other detector conditioning that must take place before starting acquisition.	Log event
T2	OBS_START_ACQ	OEvt sent just before data acquisition starts. With an optical detector this is the opening of a shutter. With IR instruments there is no shutter.	Log event. OCS samples header information.
T3	OBS_END_ACQ	OEvt send when the requested acquisition has completed. With an optical detector this occurs just after the shutter closes.	Log event. OCS samples header information.
T4	OBS_START_READOUT	OEvt indicates that the data is being transferred from the detector or other activities needed to write data.	Log event. OCS can apply the next instrument configuration.
T5	OBS_END_READOUT	OEvt indicates readout or write preparations have completed.	Log event. OCS can start a new Observe.
T6	OBS_START_DSET_WRITE	OEvt indicates that the instrument has started writing the dataset to the GDSN.	Log event. OCS can move telescope if needed.

T7	OBS_END_DSET_WRITE	OEvt indicates that the instrument has completed writing the dataset to GDSN.	Log event.

Table 18: Observe Observation Event Descriptions

Here are some notes related to writing Observation Events:

- All status values must be up to date and published before issuing the OBS_END_DSET_WRITE observation event thereby guaranteeing that all header values will be correct.
- Based on their internal architecture, builders may need to delay sending the OBS_END_DSET_WRITE observation event until it is known that all status values are correct and published.

13.5 Ancillary Files and GIAPI

Instrument studies and reviews have shown a need for instruments to add and associate non-science data, such as AO statistics, to the GDSN and GSA for the use of observers.

There isn't enough known about the requirements for this feature yet, but the following are true:

- All ancillary files will be written as FITS format files using the same software library as science data (REQ-GIAPI-DP-14).
- The content and uses for these files must be agreed upon during the instrument design phases as part of the negotiation on header content and data unit format (REQ-GIAPI-DP-15).

This interface for ancillary files is not final. The current approach is that 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 event would include the name of the ancillary file and the associated data-label.

```
postAncillaryFileEvent(String filePath, String dataLabel)
```

There may also be some standardization of ancillary file names. Here is an example call:

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

FITS is a well-documented, general purpose file format used in all areas of astronomy for a wide variety of types of data. X-Ray astronomy has developed a number of FITS formats associated with event data that are probably usable. This STSCI URL (http://archive.stsci.edu/fits/users_guide/node100.html) describes several FITS file types including X-Ray data. At this time FITS seems adequate for ancillary files.

This requirement applies only to data files that are ultimately meant for observers/PIs and the Gemini Science Archive. A diagnostic file that is not to be archived is not covered under this rule.

If needed, a builder can demonstrate why FITS is not a reasonable file format for ancillary data files before creating or adopting a non-FITS file format. These new formats would have to be documented and agreed upon during reviews prior to construction (An ICD is planned for this. ICD 16 – *Format for Documenting Public Interfaces*.) Gemini won't be producing libraries to write files in other instrument specific formats. Any code for non-FITS file formats would need to be generated by builders and factored into the software project plan.

13.6 Intermediate Files for Display

Sometimes an instrument's operation is enhanced by allowing the user to see preliminary results or specialized results. These intermediate files are created while the actions of an **observe** are underway. Intermediate files are typically temporary; they are viewed and discarded.

The GIAPI allows the builder to create intermediate files during an **observe** in FITS format and alert Gemini of their existence. The method for alerting Gemini works similarly to the ancillary file method in the previous section.

```
postIntermediateFileEvent(String filePath, String dataLabel, String hint)
```

The call takes the path to the intermediate file in the GDSN, the data-label of the science dataset currently underway, and a hint. 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:

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

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

14 Development Process

The changes for new Gemini instruments are at a development process level as well as a technical level. The first and second generation Gemini instruments were typically plagued by delays and cost overruns. Some instrument teams overspent by a factor of two on software development. Other instruments have been delivered with software defects that required significant Gemini effort to correct. Some instrument builders have allowed Gemini-delivered components to their software to become critical path items. Some instrument builders have had to start over with their software design after completely missing the boat at a key design review.

Gemini is addressing these problems with new instrument procurements by requiring greater oversight and accountability, and by budgeting significant contingency. Gemini is taking a new approach to software development that should relieve problems for both the instrument builders and Gemini. Given the large price tags on instruments, Gemini maintains high expectations for the quality of the software to be delivered with them. This section provides a brief overview of Gemini's perspective on the management philosophy and strategy for developing software for the GIAPI instruments.

The primary characteristic of the new way of developing software for Gemini instruments is its flexibility. There are fewer requirements on instrument builders than in the past, but it also requires greater responsibility. It requires that the instrument builder and Gemini be more responsive to each other than in the past. A typical Gemini instrument software system in the past would have been developed in a "sedimentary" fashion. Low-level drivers and code were developed first, building up slowly, finally being completed, with high-level software to interface with Gemini developed last. Delays along the way propagated directly into the delivery date slipping. Nothing was ready for Gemini testing or integration until the final stages of the project. The pressure was then on Gemini staff to finish integration and testing, often with significant changes required, when the pressure was high and schedules inflexible. For the GIAPI-based instruments, a more "vertical" approach is being used. Early in the project, pieces of the software will be developed and integrated end-to-end, to allow frequent releases and testing by Gemini and the builder all along the way.

This flexible and responsive system of *continuous integration* benefits the instrument builders in a number of important ways. First, it should save significant effort and money because Gemini will be providing feedback on priorities all along the way. It effectively reduces the risk to both the instrument builder and Gemini. It allows the Gemini part of the effort to be scheduled and spread out over the lifetime of the project. Since the interim releases of the software contain the full end-to-end functionality, functionality descopes can be adopted to prevent software delays from significantly impacting the overall project (integration and testing of the whole instrument can proceed, perhaps with restricted functionality, even if the software is not yet complete). It also allows software development to mature before all the hardware has been selected.

The new software development system does require some planning up-front to be effective. It requires a product backlog to be developed to define the end-user functionality that is desired. It requires some agreement on what documentation is needed, given that traditional design and ICD documents may be too rigid or updated too infrequently to guide the project over long time frames. It will require frequent communication and collaboration between Gemini and the instrument builder. We anticipate a rapid iteration time frame, with the instrument builder releasing the software to Gemini once every couple of months. The Gemini software engineers will review the release, and provide feedback on the existing software and priorities for the next couple of months' work. This rapid iteration will prevent small misunderstandings from becoming serious problems with schedule and cost implications. It is also consistent with the project management approach of frequent monthly reports and telecons/visits. It is expected that Gemini's software engineer will visit the instrument builder frequently, and not just interact with the team at the project milestone design reviews.

With proper planning and good communications, the flexibility of continuous software integration will mitigate risk and ensure that software is delivered on time and on budget that meets everyone's needs and expectations.

14.1 Iterative Development

The goal of this delopment process is to provide a more collaborative, dynamic environment for development that delivers the right software for the instrument when the instrument is delivered. The approach for development of instrument software and the GIAPI itself is to provide frequent releases of software separated by one to two months during the construction process. The iterations focus on delivery of high-level, testable functionality. As part of each release the delivered instrument software is tested by Gemini and feedback is provided to builders that influences subsequent iterations. Similarly, problems with the GIAPI exposed by builders through their development are addressed on similar time scales in the same way.

The approach Gemini is taking is based on concepts and uses terminology of the Scrum agile development process [15] and [17], but some aspects are tailored for our situation. Figure 22 is an image from [16] that shows the terms used to describe the development model.

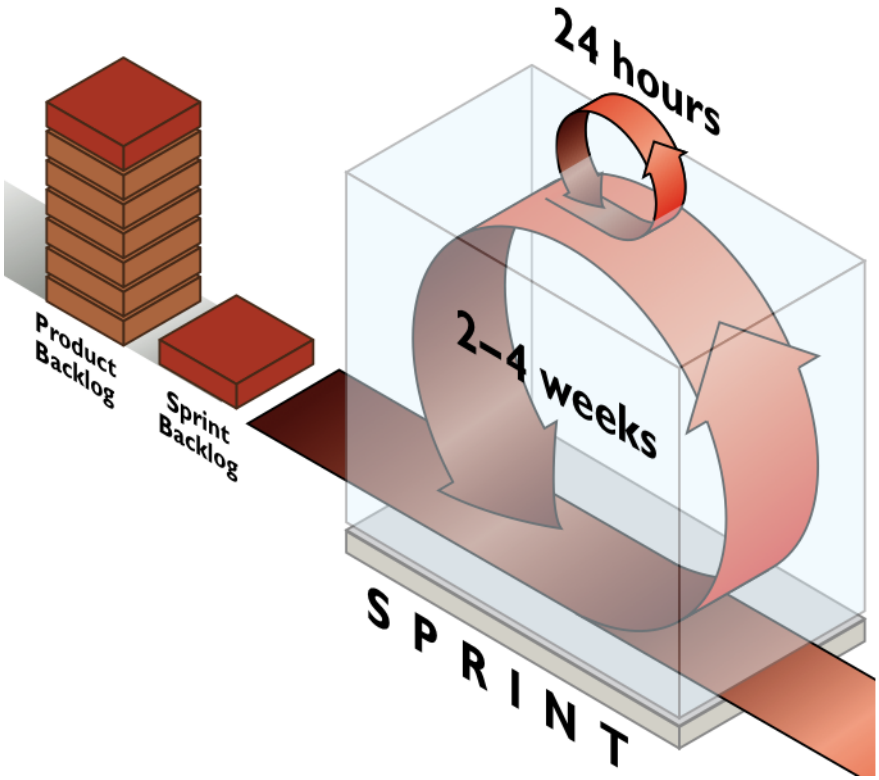


Figure 22: Scrum development process

The most important artifact of the planning process is the *Product Backlog*. This is the list of all work that must be completed; the features and functionality the software must provide based on knowledge and requirements. In Scrum, this product backlog is prepared and prioritized by the customer in the terms that make sense to the customer. At Gemini, this could be done by Gemini, but so far it is done primarily by the builder with help from Gemini. Each item on the Product Backlog is estimated by the builder programmers.

The Product Backlog and priorities are used to determine the items that go into each iteration, called *Sprints* in Scrum. Each iteration has its own backlog. The items in the sprint backlog are expanded to include all tasks needed for implementation. These are the tasks the programmers complete during the iteration.

Another diagram of the Scrum process is shown in Figure 23 from [15]. This one focuses on the regular meetings and what occurs at the conclusion of an iteration. After each iteration, the new behavior is demonstrated to the customer. At Gemini, the builder delivers the software to Gemini where it is tested (see Section 15). The conclusion of the iteration may include meetings between Gemini and the builder software developers.

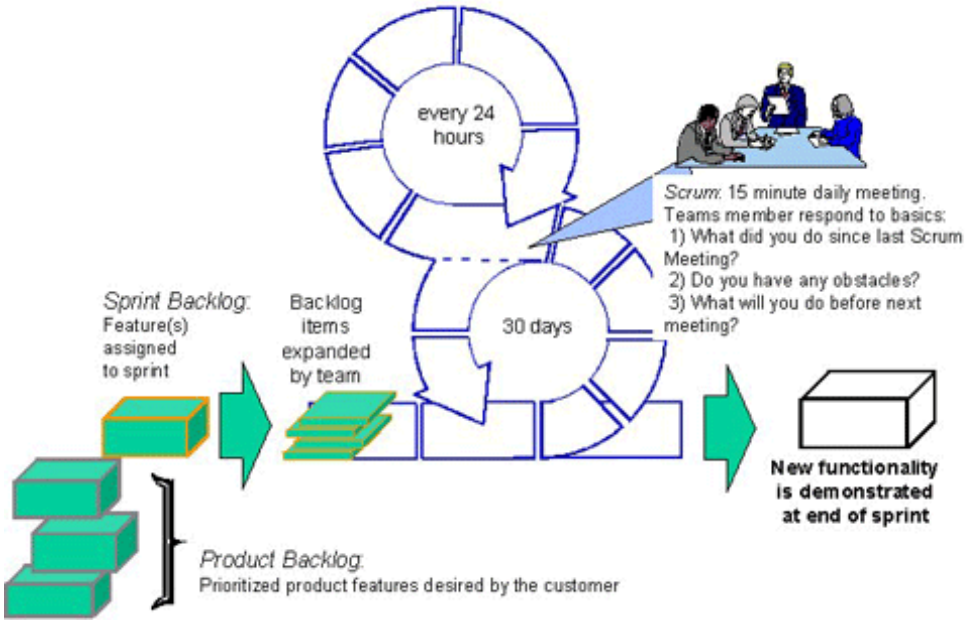


Figure 23: Another version of the Scrum process

It is a challenge to fit this development model into the sequential development model in use by most of today's astronomy projects. The easiest solution is that iterative development begins as soon as it's appropriate to start writing code. Traditionally, this occurs during the construction phase or preparations during the time between PDR and CDR. This is the easiest approach since Gemini follows the typical plan-based review process preparing for iteration-based construction.

Preparing for iterative development suggests that a generous amount of time during PDR or CDR be dedicated to outlining the Product Backlog and outlining what will occur during each iteration and release. Of course, the goal is that the content of each iteration be somewhat dynamic but demonstrating the planned approach and coordinating it with Gemini is important for reviews.

15 Testing with GIAPI

At the conclusion of each instrument release, Gemini will accept the products of the iteration and test the software using test code written by Gemini to simulate the interactions of Gemini's observing software. The results of this work will be fed back to builders.

The GMP will include a testing environment that will allow automated testing of the GMP/GIAPI and the instrument. The test suite Gemini uses will be enhanced over the lifetime of the instrument's development as new functionality is delivered. The test suite will be available to builders as well. With frequent releases and regular, rigorous testing, the problems during acceptance testing and commissioning should be eliminated. Successful completion of these tests will constitute a part of the acceptance test of the instrument software. This test suite will become the basis of the Gemini operations and maintenance instrument test suite that will be used to ensure continued proper operation of the instrument.

An important part of this approach is automation (See: Pragmatic Project Automation: http://www.pragmaticprogrammer.com/sk/auto (primarily Java, but generally good)). Most software development products today are focused on Java, and C++ is fairly under supported. But most of the continuous integration servers can build any project using make and they are flexible about running tests. Whenever code is checked into the Gemini source code repository, the continuous integration server checks the code out, builds it, and runs the tests. The tools provide reporting so you know when the tests are failing or the build fails.

16 Gemini Concurrent Instrument Development

The guidelines in this document allow streamlined development by the builder by maximizing builder software reuse and removing the dependency of the builder on major products of Gemini such as a DHS server or an

EPICS wrapper system. This approach allows the builder and Gemini to focus on their particular tasks concurrently.

The Gemini-based programmer or programmers assigned to a specific instrument (hereafter referred to as Gemini Software) have specific responsibilities during the construction process for an instrument. This section outlines concurrent development responsibilities of Gemini and instrument programmers that haven't been mentioned previously in this document.

16.1 Acceptance Test and Engineering User Interface

The instrument builder is responsible for the Acceptance Test and Engineering User Interface program as mentioned in [1]. This user interface can be executed using the approach that takes greatest value of the builder's experience and staff.

16.2 Operations Control User Interface

Gemini provides a Gemini-centric operations/engineering control and monitoring screen for use during observing that we call the Operations Control User Interface (OCUI). This user interface will primarily be a status screen but will provide control of mechanisms and devices that can be used to interactively change the configuration of the instrument during observations.

This user interface is developed by Gemini when reasonable and is not tied to the builder's instrument development schedule.

16.3 Observatory Control System Integration

Ensuring proper operation of the instrument is tied to the software test suite discussed in the previous section. All other software required to integrate the instrument into the Observatory Control System is developed by Gemini according to Gemini's schedules and should not appear in the schedule of the builder. Integration of the instruments is a collaborative effort between Gemini and the builder of course.

The OCS software components are mentioned here for completeness.

- Gemini integrates the instrument into the Phase 1 Tool.
- Gemini integrates the instrument into the Observing Tool with input from the builder. The OT contains the planning/observing configuration of the instrument.
- Gemini integrates the instrument into the OCS and seqexec. The seqexec sequences the principal systems during operations.
- Gemini integrates the instrument software with the Gemini Science Archive, Data Handling System, and Gemini Engineering Archive.

17 Tools and Releated Programming Details

Details regarding tools are present in the language glue ICDs [19][20].