Test Interferometer Control Software Design Concept

DRAFT: 2001-11-16

B.E. Glendenning (Ed.), M. Brooks, G. Chiozzi, G. Harris, R. Heald, J.Pisano, M. Pokorny, F. Stauffer

1 Acronyms

ABM	"Antenna Bus Master" A real-time computer located at the antenna that is responsible for the control and monitor of all hardware devices at the antenna. There is an identical copy of this computer at every antenna, each running identical software.
ACC	"Array Control Computer" The computer located at the central control area and responsible for coordinating all instrument activities. It is an ordinary Unix workstation.
ACE	"Adaptive Communication Environment" An open-source, object-oriented (OO), C++ framework that implements core design patterns for concurrent communication software across a range of OS platforms. (http://www.cs.wustl.edu/~schmidt/ACE.html)
ACS	"ALMA Common Software" ACS is the kernel software located between the application (on top) and other commercial and shared software over the operating systems. It supplies certain services such as messaging, logging, error and alarm handling, configuration database, etc.
ACU	"Antenna Control Unit" The system provided by the antenna manufacturer through which the antenna is monitored and controlled. The ACU primary access is through a CAN bus.
ALMA	"Atacama Large Millimeter Array" A connected interferometer telescope array expected to consist of 64 millimeter-wave antennas each 12-meters in diameter. (http://www.alma.nrao.edu)
AMB	ALMA Monitor and Control Bus (ALMA M&C Bus)
AOS	ALMA Observing System
API	"Application Programming Interface"
ARTM	"Array Real Time Machine" A real-time computer located at the central control area and responsible for the control and monitor of certain hardware (LO reference generation, fiber optic control, etc.) located only at the array center. Its function may be combined with ACC or CCC.

CCC	“Correlator Control Computer” A real-time computer located at the central control area and responsible for the control and monitor of the correlator.
CLIC	CLIC Continuum and Line Interferometer Calibration
CORBA	“Common Object Request Broker Architecture” CORBA is an emerging open distributed object computing infrastructure being standardized by OMG. CORBA automates many common network programming tasks such as object registration, location, and activation; request de-multiplexing; framing and error-handling; parameter marshalling and de-marshalling; and operation dispatching.
COTS	Commercial off the shelf
CRG	Central Reference Generator
FITS	“Flexible Image Transport Format” FITS is the data format most commonly used within the astronomy community. FITS is primarily designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images or 3-D data cubes) and 2-dimensional tables containing rows and columns of data. (http://fits.gsfc.nasa.gov/)
GPIB	General Purpose Interface Bus
GPS	Global Positioning System
GBT	Green Bank Telescope
HLA	High Level Analysis
IDL	“Interface Definition Language” IDL is part of the CORBA standard and permits interfaces to objects to be defined independent of an object’s implementation. IDL is used as input to an IDL compiler that produces source code that can be compiled and linked with an object implementation and its clients.
IRAM	"Institut de RadioAstronomie Millimétrique" (French) IRAM and other agencies are building ALMA. (http://iram.fr/)
LCU	Local Control Unit (for ALMA, a VME crate)
M&C or MC	Monitor and Control
NTP	Network Time Protocol
ODBC	Open Database Connectivity (TBC)
OTC	Optical Telescope Controller
PPC	“PowerPC” A range of processors developed by an alliance of Apple, IBM and Motorola. For ALMA it is the processor embedded on a SBC.
PPS	Pulse Per Second (often 1PPS)
PTC	Pointing Computer
QoS	Quality of Service
RDBMS	Relational Database Management System
TAI	“International Atomic Time” TAI is a laboratory timescale. Though TAI shares the same second as UTC, UTC noticeably separates the two

	timescales in epoch because of the build-up of leap seconds. At the time of this writing UTC lags about half a minute behind TAI.
TBC	To Be Confirmed
TBD	To Be Determined
TICS	Test Interferometer Control Software
UML	"Unified Modeling Language" The UML is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.
UTC	"Universal Time Coordinated" UTC is the basis of civil timekeeping. Most time zones differ from UTC by an integer number of hours, though a few differ by n+0.5 hours. The UTC second is the same as the TAI second. In the long term, UTC keeps in step with the Sun. It does so even though the Earth's rotation is slightly variable by occasionally introducing a leap second.
UVW	Coordinates of an interferometer sample in the Fourier plane.
VLA	Very Large Array (http://www.aoc.nrao.edu/vla/)
VLT	"Very Large Telescope" (http://www.eso.org/projects/vlt/)
VME	Versa Module Europa." VME is a computer backplane bus system that makes use of the Eurocard standard. It is defined by the IEEE 1014-1987 standard.

2 Introduction and Summary

Brian Glendenning
Last Updated: 2001-10-05

This document describes the design concept of the control software for the ALMA test interferometer. The test interferometer consists of the initial two antennas from the European, Japanese, and US antenna vendors along with the associated first generation hardware and software. It will be located at the VLA site.

While much of the test interferometer software may be reused for the prototype interferometer (*i.e.,* at the VLA site but with the next-generation (prototype) correlator and electronics) and final ALMA array, that design will be the subject of follow-on documents.

Table 2-1 Different ALMA Development Stages

What	***Where***	***Major Features***
Test interferometer	VLA	Prototype antennas, test correlator, evaluation receivers
Prototype interferometer	VLA	Prototype antennas, prototype correlator, prototype receivers
ALMA Array	Chile	Production antennas, production electronics, baseline and future correlator.

For the test interferometer, we must be in a position to carry out the tests required to evaluate the antenna and other hardware. Thus, our principle goal is to have sufficient software to enable the commissioning plan to be carried out. It must be emphasized that while astronomical observations will be required to evaluate the hardware, astronomy *per se* is not a goal of the test system.

In general, we would describe this design as conventional in the sense that designs like it have been implemented for other telescope control systems (*e.g.,* the GBT and VLT). There are no drivers that we know of that would require us to come up with a radically new design. We document our design using UML, an industry-standard object-oriented design language. We rely on CORBA to provide the required remote execution and coordination facilities.

This design should of course be validated against our requirements, which are described by Brooks *et. al.* (2001). Fundamentally the role of the control software is to take high-level observing commands from scripts and GUIs, to turn these high-level commands into detailed control of the test interferometer, finally producing monitor archive data for engineering data analysis, and "science" data files (typically in FITS format) to be used for holographic and astronomical calibration of the instrument. In brief, the scope of the control software is "input script to FITS." In particular, data processing is not an element of the control software.

Almost all devices will be attached to a CAN bus operating in a master/slave (polled) fashion. The bus will operate at 1Mbps and is capable of at least 2000 polled operations per second (up to 8 bytes of data per transaction). Devices on the CAN bus will be responsible for implementing a simple in-house protocol to map CAN message IDs to internal device addresses (Brooks and D'Addario, 2001). A few devices will have other connections, in particular Ethernet and GPIB. CORBA will be capable of at least 1000 remote method invocations per second.[1]

There are multiple real-time computers in the system. The CAN buses are attached to (mastered by) a PPC VME based computer running the VxWorks real-time operating system. There is one ALMA PPC system at each antenna, and two at the center. One central system is used to master the central CAN devices. The other system is embedded within the test correlator, which does not have any CAN bus.

Centrally there is a general-purpose computer running Linux that is the overall master for the system, which is known as the Array Control Computer (ACC). There are also some ancillary systems in the center for, *e.g.,* operators to sit at.

The antenna-based systems are connected to the central systems via point-to-point Gigabit Ethernet network connections. COTS solutions for Gigabit Ethernet are available that support 40km fiber runs and quality of service (QoS) guarantees. The central computers are interconnected with 100Mb switched Ethernet on a network segregated from the rest of the networks at the VLA.

Logically, the software is partitioned so that control flows in a master-slave fashion from a central executive who controls high-level ("composite") software devices that in turn control their constituent parts. The lowest level software devices are referred to as device controllers, and represent a proxy for the actual hardware – that is, they communicate with the hardware. Data – monitor and backend – is collected from the devices by a collecting process in the

[1] Fewer CORBA messages than CAN messages are needed because high level commands will be turned into several control point accesses, and because monitor points are buffered in the ABM.

real-time computer attached to the hardware, and buffered up for distribution via a publish/subscribe mechanism to consumers of the data, which include the processes that format and archive the data. The software is distributed amongst computers so that only the device controllers and software directly concerned with low-level device activities are on the local real-time computer. Higher-level software entities are concentrated on the ACC.

The entire suite of software required for normal observations is called the ALMA Observing System (AOS). This encompasses more than just the control software. For example, it includes the software required to calculate various calibrations needed for routine operation of the array, such as pointing calibrations. The "executive" is the software process that initiates and provides for high-level oversight between the major subsystems of AOS. For the test interferometer, these are principally the control software and telescope calibration software.

The principal observing input interface to the system will be with commands in the Python scripting language. Typically, these commands will be prepared as scripts in advance, but they may also be entered interactively for debugging or other purposes. The command primitives will be expressed as an IDL API, which will be bound to Python (and other languages with an IDL binding). Another primary human input for the system will be source catalogs and ephemeris files, which will be defined in a TBD ASCII format. Some useful GUI displays shall be available during operations – for example showing the status of the cable wraps. It is TBD whether the GUIs will be display only, or will allow for some control functions.

Engineering access to devices that are installed on the test interferometer will be implemented via access to the device controller interface or to the I/O routines directly from the engineering workstation.

The ALMA Time System will establish synchronized switching cycles and mode changes, and it will provide time-stamping for the resulting measurements. This must be done across the entire instrument including the central building and the geographically dispersed antennas. Additionally, the Time System must be accurately related to external measures of time to correctly determine the position of astronomical objects of interest.

The master clock will be implemented in the central real-time computer by counting 48ms timing events from the central reference generator that has been tied to external time systems with a GPS receiver. The fundamental time system of the interferometer maintained centrally is known as array time. It is tied to TAI at initialization time, and drifts only very slowly thereafter.

Although TAI is the fundamental external time system used by the software, other time systems such as UTC and local sidereal will be accepted from user input routines.

Most devices do not have precise timing requirements. For those that do, the control software must arrange to have monitor and control commands sent to the device in precisely defined windows in the pervasive 48ms timing period. This is accomplished by sending time-tagged commands from the center sufficiently in advance of when they are required to account for the non-determinism in the network and general-purpose ACC. The commands are then staged in the real-time computers until they are required at the hardware.

Slave clocks (in the other real-time computers) are tied to the master clock by being given the array time of a particular timing event. They maintain time thereafter by counting 48ms timing events.

There are several characteristic ("looping") timescales of interest to the control software:

- 2ms: This is the shortest timescale at which any device will require interaction (the total power detectors). Timescales faster than this are always handled by hardware.
- 48ms: This is the period of the pervasive timing event sent to all hardware with precise timing requirements.
- 70ms: Fastest correlator dump time.
- 1s: This is the fastest timescale for observational changes, e.g., source change or change in correlator setup.
- >1s: Most devices will be monitored or controlled at rates slower than 1Hz, often much slower (300s).

Some model values will be required for some devices – for example, the test correlator will need a delay model. These calculations will be encapsulated in "model servers" running centrally. The central control software will call these servers and send a parameterized result (e.g., polynomial as a function of time) to the device drivers that need them. The device drivers might in turn evaluate these parameterized models at a faster rate and send the results over the CAN bus to the underlying hardware.

A particularly important device is the test correlator. It is fundamentally based on the design of one quadrant of the GBT spectrometer, modified to provide delays and cross-correlation capabilities. It was also modified to enable data dumps to be (optionally) synchronized to the pervasive ALMA 48ms pulse. The internal design is quite complicated and is described by Pisano (2001). As a device, it behaves like all other devices as a slave, accepting relatively high-level configuration and control commands, and emitting the auto and cross-correlation data. Unlike other devices, the control software interacts with this device over Ethernet.

Monitor data is archived in an RDBMS with an ODBC interface for convenient access from commercial packages, and the science/calibration data is formatted in a single FITS format (except for the optical telescope data which uses a format defined by the TPOINT pointing analysis software). The FITS files are written directly to disk rather than into some more elaborate archive. Production of the science data requires the gathering together both monitor and backend data, calculation of some derived values (e.g., UVW), and flagging.

As the science data becomes incrementally available, the telescope calibration subsystem is notified that more data is available to be read. The telescope calibration software is based on the IRAM CLIC package and is external to the control software, although they must be able to communicate with each other. CLIC will write calibrations that are needed by the control software into a file that is imported as needed by the control software

The ALMA Common Software (ACS) group will provide much of the underlying functionality needed by the control system, such as the logging, error, and alarm subsystems.

3 Physical Architecture

B. Glendenning
Last Updated: 2001-11-16

As shown in Figure 1 the computers and networks break down into antenna-based systems and central systems.

At each antenna there is an ABM – a VME Power PC based VxWorks computer. Its principal role is to provide real-time control of the devices at the antenna based upon infrequent time-tagged commands from the center. All devices with computer interfaces are attached to a CAN bus. More details about the properties of these interfaces are described in section 10.4.

Data transmission from the total power detectors are carried on a separate CAN bus to preclude the possibility of overloading the general M&C bus with data transmission. Those devices, however, are controlled over the regular bus – hence they will have two CAN connections. The control bus is referred to as the AMB (ALMA Monitor and Control Bus) to distinguish it from CAN buses used for data transmission or other purposes (in particular, within the prototype and baseline correlator).

A particularly important device is the Antenna Control Unit (ACU) that is provided by the antenna vendor. It is implemented as a real-time computer running the VxWorks operating system[2]. Like any other device it is principally controlled over the CAN bus, although it also has an Ethernet interface for software maintenance and the setting of static parameters that do not need to be changed in normal operations. The Vertex antenna also has a Pointing Computer (PTC), supplied by the vendor. The PTC is used for the pointing model, subreflector control, and other miscellaneous functions. An Ethernet switch is used to route traffic to all devices on the local network, and to convert between fiber and twisted pair. The switch can also implement QoS if desired (for example, it could be used to prioritize traffic between the ABM and ACC).

The optical telescope is also commanded via the CAN (AMB) bus. It has an analog video output which transfers the signal from the telescope independently of the control system to an Optical Telescope Controller (OTC) which processes the video signal and derives centroid parameters for the target star.

Each ABM is connected to the central systems via a point-to-point Gb Ethernet network which terminates at a central switch. Up to 40km fiber runs are possible using available COTS equipment. While the antennas are much closer than this for the test interferometer, a long spool of fiber may be inserted to simulate the long cable runs for the ALMA array. The switch is in turn connected to a 10/100 Mbps switched Ethernet on which all central ALMA computer systems required to operate the test interferometer are attached. This switched network is in turn connected to the rest of the VLA networks, and ultimately the general Internet, through a hub.

At the center there are two real-time computers. The Array Real-Time Machine (ARTM) plays the role of the ABM, providing local real-time control of its attached CAN devices. In addition, for control of two COTS synthesizer devices it will master a GPIB to which they are attached. It will also be attached to a GPS receiver which, along with the Central Reference Generator (CRG), will allow it to be the fundamental source for array time.

The Correlator Control Computer (CCC) is the other central real-time computer. It provides the interface for the test correlator, and provides detailed control of the correlator hardware. The software in the CCC is described in more detail in section 11.10. Both the ARTM and CCC are VME/PPC/VxWorks based systems. The CCC communicates with the correlator hardware via an RS-232 connection.

The Array Control Computer (ACC) plays a central role in the test interferometer. It is responsible for controlling all hardware in the array (indirectly through the ABM, ARTM, and CCC computers) under the command of high-level observing scripts. It also executes ancillary software including model (*e.g.,* delay) servers, and data formatting. It is a high-end x86 based workstation running the Linux operating system. If necessary for performance reasons, the ACC functions could readily be split into multiple computers.

---

[2] The Vertex antenna has a CompactPCI with an x86 processor. The EIE antenna has a VME/PPC based system.

There will be a few general purpose Linux and Windows systems on the switched network for operator access, engineering and astronomical data analysis, software development, and the like.

For meteorological data the test interferometer will have its own weather station. Details of this connection are TBD, but the data will likely be transferred via Ethernet from a local controller PC which has a serial connection to the meteorological eqipment.

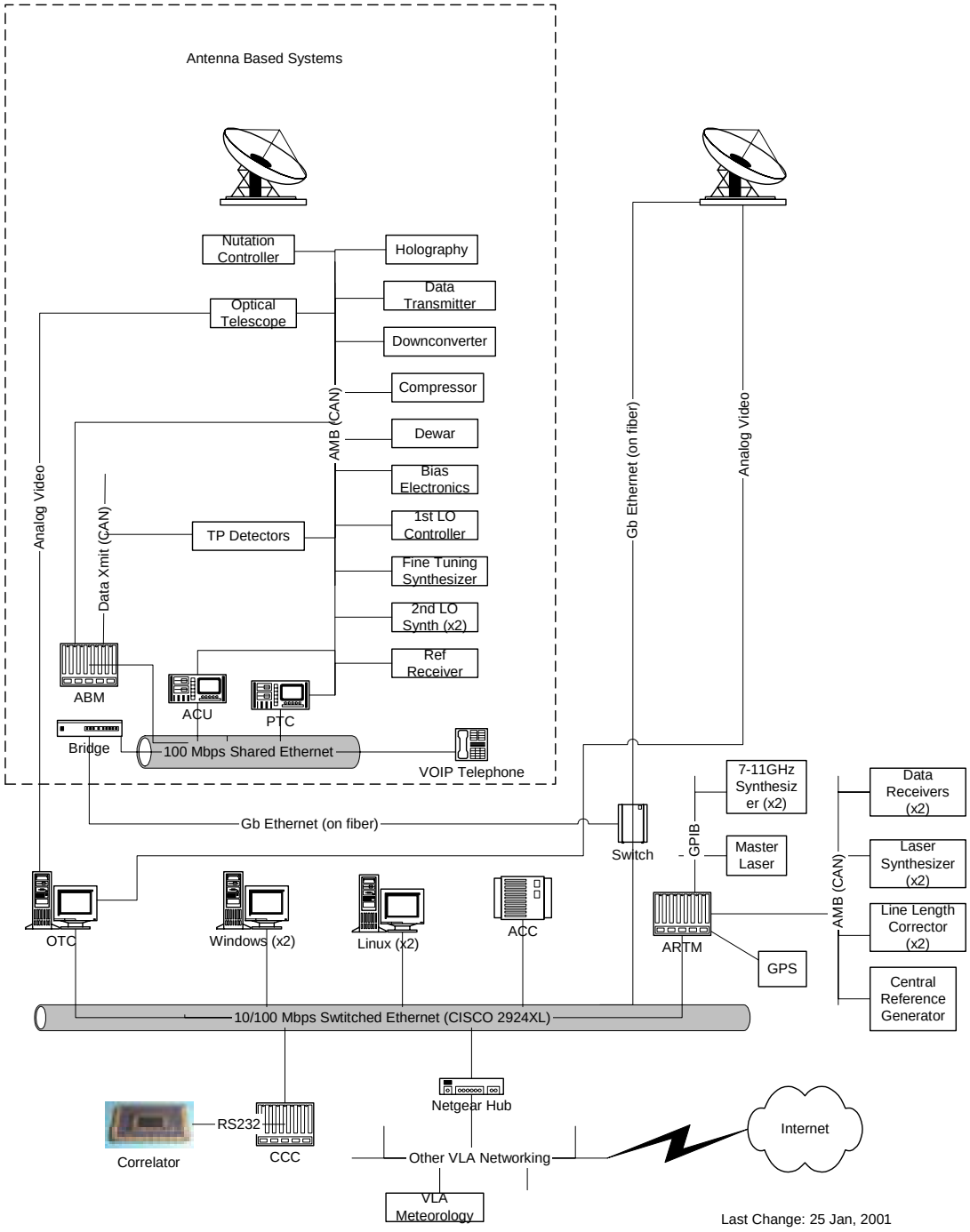


Figure 1 - Physical architecture for the test interferometer. Only the Vertex antenna has a PTC.

4 Test Interferometer Logical Architecture

G. Harris
Last Changed: 2001-05-05

The Test Interferometer [TI] is controlled by software similar to the final instrument, but in a simplified form. The following diagrams indicate the organization of the TI Control Software

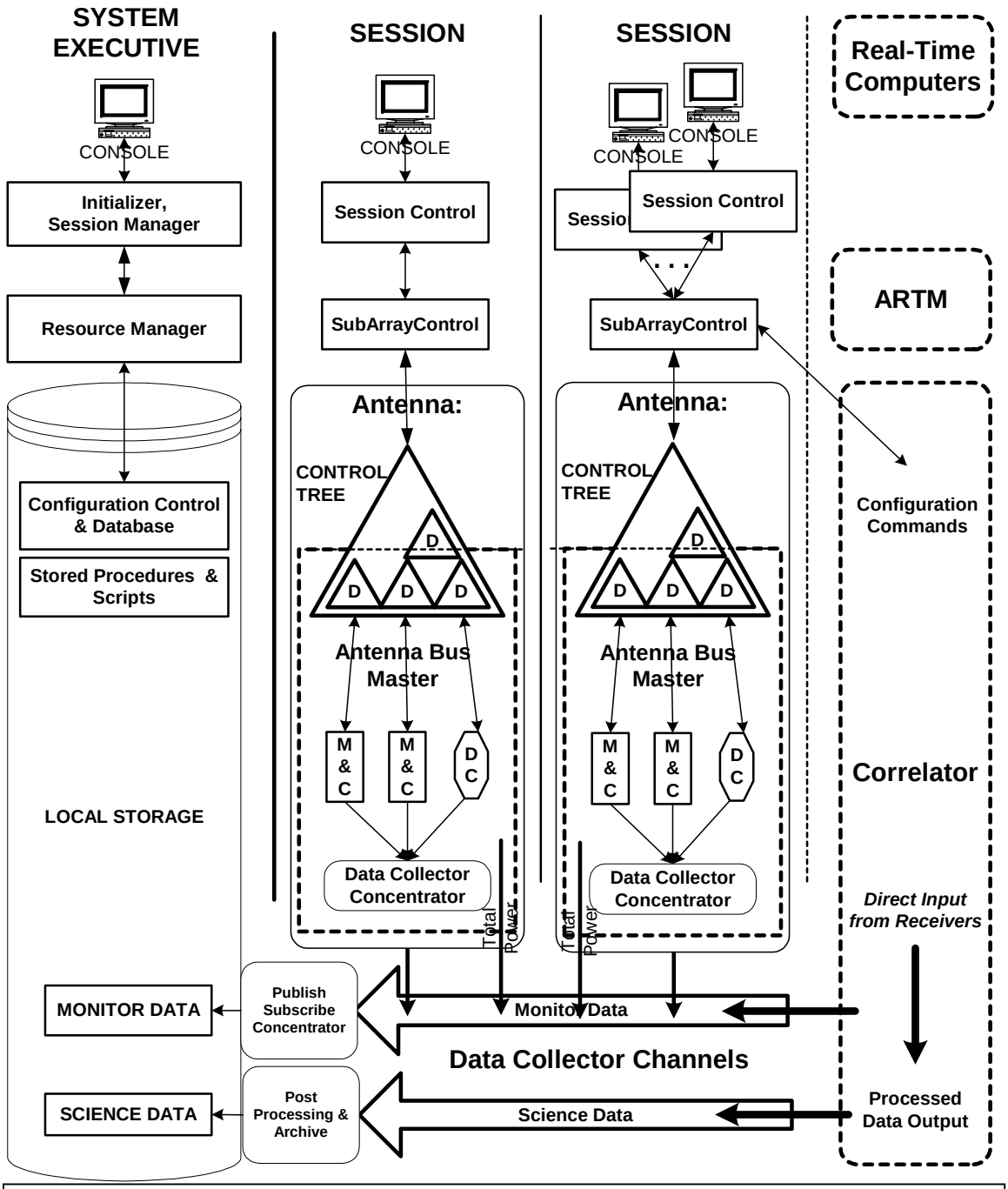


Figure 2 - TI Logical Architecture: Single Dish

[TICS]. They show the functional organization of the physical parts described previously in the section on physical architecture.

The TICS software must:

- manipulate each antenna separately in simultaneous single dish sessions.
- operate both antennas together in a two antenna array with the correlator.
- exercise each device and collect data to evaluate its behavior.
- provide for interactive use and the execution of stored procedures.

SYSTEM EXECUTIVE: [see figure] Although described in more detail in the section on the executive, we should note here that this section stores the configuration of the instrument, retrieves it at startup and initializes the each Device. It then creates a session for the user, allocates antennas and possibly the correlator to it. It also starts any data collection procedures to provide a way for the tester to capture data from a Device and starts any other system services. The exec contains various parts such as session manager, resource manager, configuration database, stored procedures and other parts described later in this document.

SESSION: [see figure] A session is an operating system process, created by the exec, which actually performs commands and operates the hardware: antenna, receivers, correlator, etc. Using a session, the user enters commands or retrieves stored commands on the system if in interactive mode. If in automatic mode, prepared observing scripts may be executed. The part executing commands is called the Session Control.

Commands are propagated from a Session Control to the SubArrayControl and then to each antenna through a control tree, a software construct representing each Device on the antenna. Notice [as shown in the Single Dish diagram] that several Session Controls may access a SubArrayControl in the Test Interferometer. Subarrays are initialized when the first session attaches and cleaned up when the last session detaches.

The method by which commands propagate is called CORBA, an industry standard. Use of distributed object oriented technology is quite extensive in this instrument. Every possible piece of the software is object based, from the configuration data, to the scripts and their functions, to each Device in the control tree and even to the hardware device control points on the antennas. This facilitates the creation, management and execution of the control software. When we use the word "Device" with a capital letter, we indicate the software proxy for a physical hardware device.

Some Devices are simple and others are treated as a group. The letter D inside the control tree triangle represents a Device, perhaps a composite hierarchy of parts such as LO, filters, etc. The hierarchy of Devices, represented by the triangle, expands commands as they propagate through the control tree. A simple command by the user results in many commands to the bottom level Devices.

Some software Devices are in the central computers, in what may be called the logical time part of the system. Other Devices are in real-time computers such as the CCC, the ARTM, or even out on the antennas, distributed next to their associated physical devices in the ABM**. These external computers are represented by dotted lines and operate in real time.** The dotted line in the control tree of an antenna represents the transition from logical time to real time as communications move from the logical time section to the real time section.

Once commands reach the ABM, they are formatted appropriately and routed to appropriate M&C points. Monitor points read the value of a single hardware access point and Control points can set [write] the value of a single hardware access point.

Sometimes a behavior is needed in real time which physical devices do not have. Rather than trying to perform this more complex behavior remotely from high in the control tree, a software construct called a Device Controller [DC] is used. For example, a nutator sequence may be given as a single command from the command tree resulting in many actions by a Device Controller in the ABM, all synchronized in time.

This architecture decouples the control system from the data recording. The control system is not dependent on the data monitoring and recording system. Or vice versa. Equipment may be monitored even when not being used for observations. Just as the antennas are configured and initialized before giving them to a work session, so also is the data collecting system turned on and initialized by the system executive. Other subsystems and services are also decoupled as much as possible.

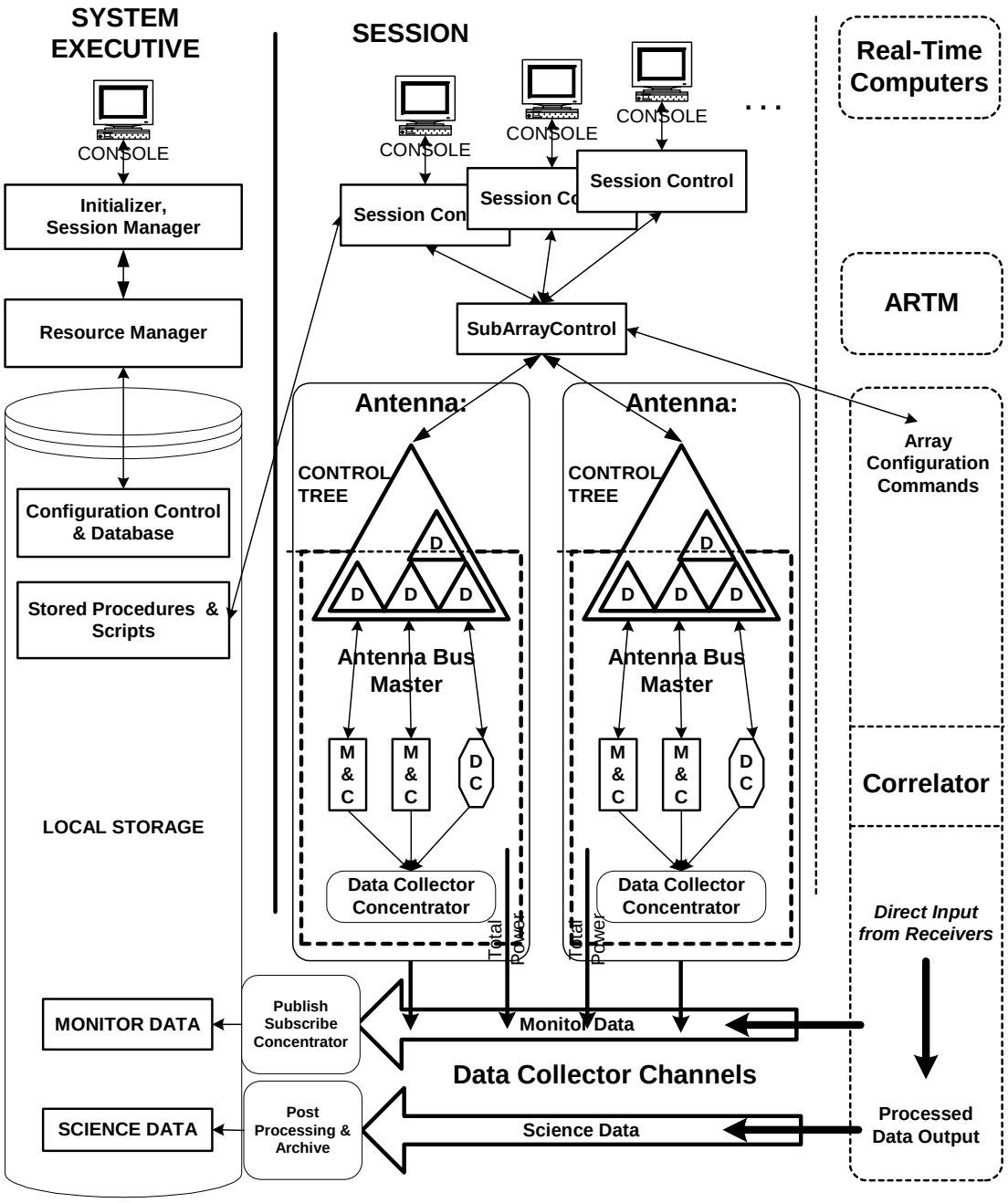


Figure 3 - TI Logical Architecture: Interferometer

When operating in interferometric mode [see interferometer figure], a single control drives both antennas - constituting a sub-array. The SubArrayControl distributes the current command from Session Control to the antennas, correlator, etc.. Generally all antennas

receive the same command, but each can also be individually addressed. Any per antenna differences in processing a command are handled by each antenna's control tree.

Another independent computer with support electronics is the ARTM. It coordinates with external clock sources, internal frequency generators and other equipment to provide a timing standard to the array. It also is responsible for the control of central devices, excepting the correlator.

The correlator is accessed on two levels. First it is configured on an array basis and commanded as shown in the drawing by the SubArrayControl at the subarray level As data coming from the antennas is sent to the correlator, it is processed and the correlator output goes into the science data.

Since each antenna is given to an independent session in single dish mode and both are given to a single session in interferometric mode some allocation control is needed. Although some of the Resource Manager must be implemented to initialize the TI, allocation will be handled by a manual action at the system executive level.

A description of the operation of the sessions and the exec which controls them is found in the following section, together with a simplified class diagram.

4.1 Design Notes

1. Notification and other top-level services must be started first.
2. Configuration database is centralized. For ACS 1.0 local databases are derived from the central database.
3. Devices bootstrap themselves by reading their configuration data from a database interface, provided as start of the initialization process.

5 ALMA Common Software

G. Chiozzi

Last Changed: 2001-05-04

5.1 Overview

The ALMA Common Software (ACS) is located in between the TICS Control Software and other basic commercial or open source software on top of the operating systems. It provides basic software services common to the various applications (like antenna control, correlator software, data pipelining).

ACS is designed to offer a clear path for the implementation of applications, with the goal of obtaining implicit conformity to design standards.

Initially the main users of ACS will be the developers of the TICS software. The generic tools and GUIs provided by ACS to access logs, Configuration Database, active objects and other components of the system will be also used by operators and maintenance staff to perform routine maintenance operations.

ACS 1.0 will be used on one side to provide support to TICS development, similar to that which is to be used later for ALMA. On the other side this experience will provide feedback from a realistic sized project to ACS development.

5.2 Technologies

At the very core of ACS is CORBA. The reasons for using CORBA are in short: Object Orientation, support for distributed systems, platform independence, it is a communication standard, and it provides a variety of services.

The ALMA software will have to be as much as possible independent from the operating system and will actually run on multiple platforms (Linux and other flavors of UNIX and VxWorks). We have therefore decided to select the Adaptive Communication Environment (ACE) as the basic multi-platform library for software that must be portable. This package provides portable operating system interface services and implements a wide set of classes specifically designed for the implementation of distributed real-time systems.

ACE is also the core of The ACE ORB (TAO), a high performance real-time CORBA implementation.

The Object Model for ACS is based on the concept of Distributed Object, which identifies three entities:

- **Distributed Object -** Instances of classes identified at design level in the ALMA system, with which other components of the system interact, are implemented as Distributed Objects. In particular, at control system level, Distributed Object is the base class used for the representation of any physical (a temperature sensor, a motor) or logical device in the control system.
- **Property** - Each Distributed Object has a number of Properties that are monitored and controlled (status, position, velocity, electric current).
- **Characteristic** - Static data associated with a Distributed Object or with a Property, including meta-data such as description, type and dimensions, and other data such as *units, range* or *resolution.*

5.3 Services

ACS 1.0 provides the following services, used in TICS by higher-level components of the Control System:

Distributed Object: core implementation for Distributed Objects, Properties and Characteristics that are at the base of the logical model.

Distributed Object Life Cycle: management of Distribute Object instantiation and destruction, object lookup (based on CORBA Naming Service), start-up and shutdown of services

Configuration Database: Distributed Objects bootstrap themselves reading configuration information from the configuration database.

Data Channel: implementation of a data pipe, based on CORBA Notification Service, to transfer efficiently continuous flows of data. It is used by many of the higher level services.

Event System: implementation of data retrieval by event, monitors and periodic timers.

Logging System: API for logging of data, actions and events. Transport of logs from the producer to the central archive. Tools for browsing logs.

Monitoring System: Monitoring and archiving of system and engineering values on a fixed periodic basis or on the occurrence of specific changes in the value.

Error System: API for handling and logging run-time errors, tools for defining error conditions, tools for browsing and analyzing run-time errors.

Time System: Time and synchronization services.

Alarm System: API for requesting notification of alarms at the application level.

Sampling: low-level engine and high-level tools for fast data sampling (virtual oscilloscope).

Some of these components are only partially implemented in ACS 1.0 or are in the form of prototype. Some others are actually implemented as part of TICS development and integrated in ACS.

For more details, refer to the ACS Architecture document and to the design documents for ACS 1.0. (TODO – insert actual reference).

6 AOS / Executive

G. Harris
Last Changed: 2001-05-05

The ALMA instrument has a executive supervisory program called the exec The exec initializes the instrument, creates processes for users called sessions, allocates resources, and launches services which collect and process data. It also supports operator activities. Some of its main parts are diagrammed in the figure.

The action of the exec is similar to initializing and running an operating system:

- During startup, a resource manager locates and initializes the physical hardware devices, using information from a configuration control database as needed.
- System processes such as services are started and necessary communications connections established (e.g., the CAN discovery and monitoring processes).
- Software Devices are created by a resource manager through a download procedure as proxies for hardware devices. These access the physical devices.
- These Devices are registered in a component naming system (the CORBA Naming service, through the ACS Manager), making them visible and available to processes in the system.

- Antennas are constructed from Devices and allocated to a resource pool, along with the correlator and other resources.

- User processes, called sessions, are started to use the resources and perform work.

Exec parts which perform these functions are shown in the diagram in dotted boxes as follows:

- Run-Time Control - Activities which create and operate sessions for users:

    - Resource Manager - initializes and instantiates Device objects from hardware, constructs complex Devices from simple Devices, allocates resources for use, re-initializes and reallocates after use. This is described in more detail in [TODO - Device Creation Reference].

    - Session Manager - creates and terminates sessions, performing associated housekeeping.

- Services - Activities which other parts of the exec use in common:

    - Services Manager - starts and stops each independent service task comprising the exec.

    - Event Manager - Provides underlying event dispatch services, for example: data collection and distribution using the publish/subscribe mechanism.

    - Object Manager - starts and stops CORBA services, including name and ID services as well as providing IDL interfaces.

    - Configuration Control & Database - a service providing reference information for other services and processes. It retains initial and current Device configurations, persistent information on a per-session basis, and some system-wide persistent information.

    - Service initialization may interact with the ACS Manager – TBD.

- Initialization and Shutdown

    - Initializer - starts and stops the exec and its subsystems.

    - Loader - starts and stops basic communications which other services use, downloading when requested. (TODO – clarify meaning).

Session activities use the following parts:

- Session - An operating system provided process which is used for executing commands and performing other processing activities.

    - Session Control - The top-level procedures driving the session activities. In the TI, Python scripts run in an interpreter, executing manual input and stored files. It issues commands to the SubArray Control.

- SubArray Control - A communications process created by the Resource Manager. It communicates commands to the devices comprising the subarray. In the TI, the Resource Manager will retain ownership of the subarray rather than passing ownership to the session. The Resource Manager will track how many sessions are using a particular subarray - initializing for the first session attaching to the subarray and cleaning up after the last session detaching from the subarray.

    - Correlator - A computational device taking input from the antennas and processing it into science data to be stored as output. There is only one.

    - Antenna - The subarray may have one or more antennas. Each is composed of all devices: mount, receivers, cryo, etc.

There may be one or more sessions operating simultaneously. Some sessions may be used by the operator for maintenance and configuration activity. Others will be used for test and observation activity.

Testing will use either independent sessions with a single dish each and possibly the correlator - or one interferometric session with all resources. In the test instrument more than one session control talking to a SubArrayControl may be allowed.

Using the class diagram and the architecture diagram from the preceding section, we can see how this will operate:

1. The operator will start the AOS/Executive after operating system startup.

2. The exec will first start the resource manager. The resource manager will be responsible for using the configuration database and other control components such as persistent objects to initialize the antennas as described in [TODO – insert Device Creation xref].

3. Each antenna will be completely initialized, including the control tree and ABM, and allocated as a unit to a sub array.

4. During this process all devices in the antenna will be registered in the object naming system.

5. Once these devices can be identified, the exec can also start the data collection process in each antenna's ABM computer. The ABM equipment monitor data collector concentrates information from about a thousand points per antenna. There is also a central collector which gathers the data from each antenna's collector in a fan-in dataflow. This is called a publish/subscribe mechanism. Once the data is collected into the central ACC, other processes may subscribe collected data from this central collector now also a publisher (e.g., Monitor point archiving and Data Production, each described later). An alternative ACS logging mechanism is available for applications that want to subscribe to monitor points that are not “blocked” as described here.

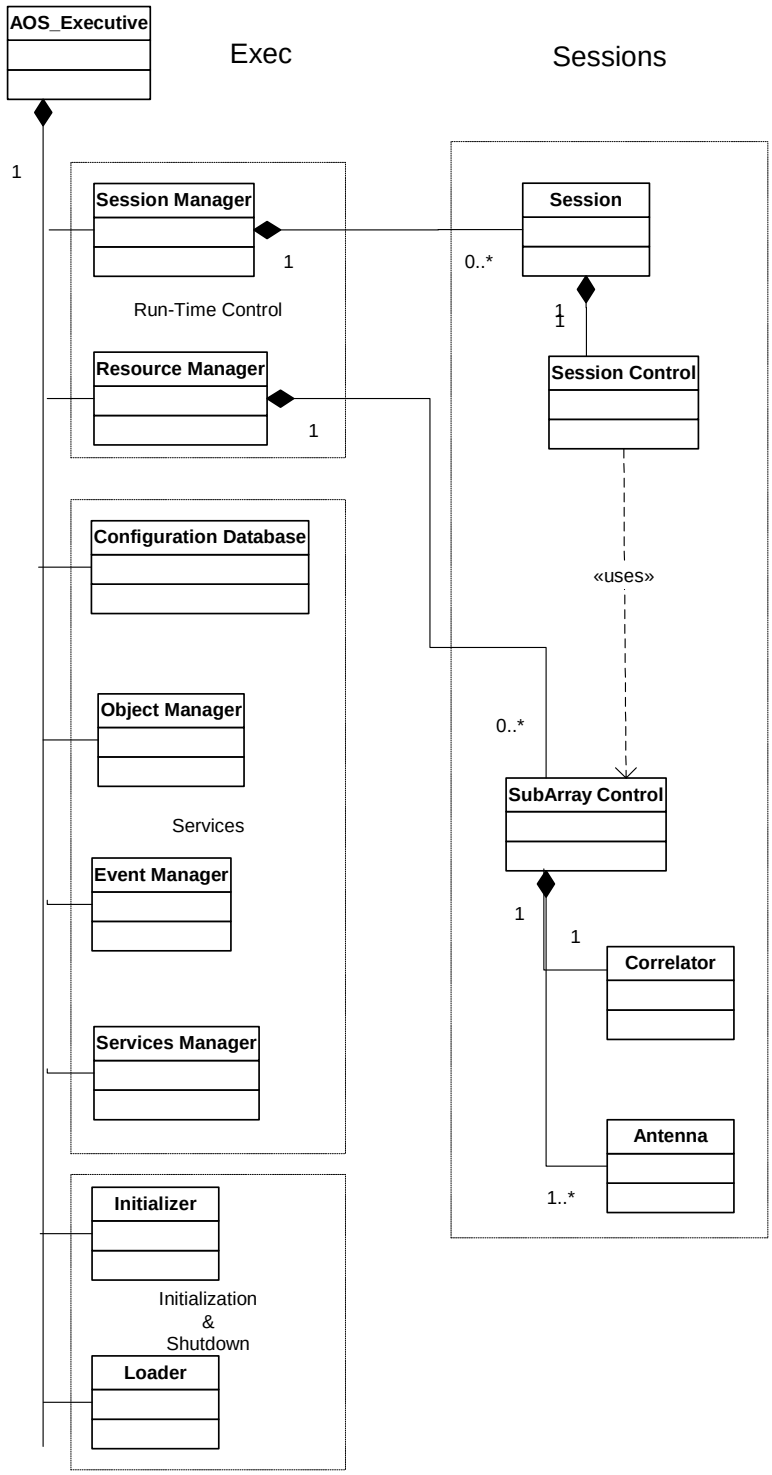
6. With all antennas initialized and the data collector running, the executive can start sessions for the users and give them appropriate resources depending on the desired observing mode: independent single dish sessions or one interferometer session.

7. When the last session using a resource terminates, the exec will clean up the resources and possibly re-initialize them or place them in a known state for another session.

Operator sessions will probably be necessary for various standard activities, configuration database maintenance, antenna parking or stowing, shutdown, etc.

Most of these utilities will be Python scripts executable as shell commands or executed from a running Python session.

Figure 4 - Test Interferometer Control Software



7 Data Channel

F. Stauffer
Last Changed: 2001-10-31

The data channel is a general mechanism to asynchronously pass information. In particular, the following features are needed:

- Publish-subscribe ("push")
- Asynchronous messages
- Filter messages by attributes
- Buffer messages

The CORBA Notification service provides the features needed and is the model used. It provides a notification service that users can connect to as publishers or subscribers. A notification service can run on any of the computers and be distributed to remote users with CORBA. The service can handle 100 to 500 messages per second depending on the level of QoS used.

The following figure is a publish-subscribe example. The data channel is used to move events, which can have data, between publishers and subscribers. This is a model of moving information from one or more publishers to many users with a concentrator to insulate the original publishers from the users of the data. For example, the monitor data or logging from the real-time computers would be moved to concentrators on the ACC that would pass the data to the outside users. Outside users can filter for the data they are interested in.

The Data Channel (Notification Service) is a supported component of ACS.

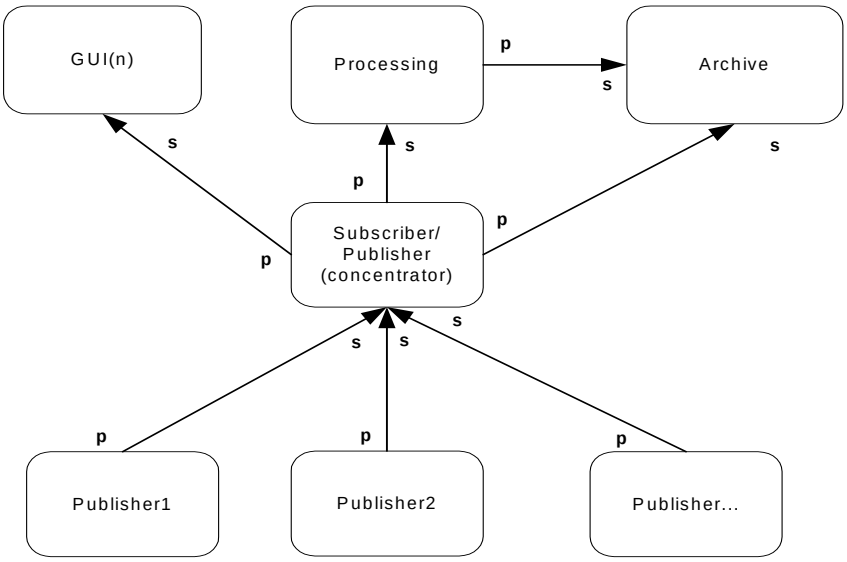


Figure 5 - Data channel with publishers and subscribers built on CORBA notification service. The lower case p, s in the diagram shows the role of the connections. This is just an example of how a data channel can be used.

8 Configuration Database

G. Chiozzi

Last Changed: 2001-05-05

The Configuration Database is used to centralize and store information to configure the TICS Distributed and other objects. TICS has several different software domains – hardware control, observing software, data production, and miscellaneous services. Each domain has its particular needs for a configuration database.

The database requirements are:

- Hierarchical
- Interface to the TICS
- Interface to the users

The ACS 1.0 Configuration Database is used for TICS. Later ACS releases will replace the database engine, probably with an XML based system.

8.1 Definition of the Configuration Database

The ACS 1.0 Configuration Database is defined using the `dbl` (database loader) syntax:

- The structure of the hierarchical database is defined by instantiating `points`
- `Points` are defined as collections of `Attributes`, where an `attribute` can be:
- another `point` (that is a node in the hierarchical structure)
- a `primitive type` (a leaf in the hierarchical structure), containing values of integer, floating point, strings and other primitive type
- `Points` can be
  - Defined directly at instantiation time as collections of `Attributes`
  - Instances of `Database Classes`
- `Database Classes`
  - Are hierarchically defined as collections of `Attributes`
  - Inheritance can be used to extend and modify Database Class definitions

As a general approach, part of the implementation of each Distributed Object is the definition of a corresponding Database Class for the configuration database, that provides all default values for Properties and Characteristics.

The configuration database will contain as needed points instantiating the DO class and overwriting as necessary the default values.

As a baseline implementation, dbl configuration files will be edited and maintained with a text editor.

The possibility of defining a higher level XML format for the definition of the configuration database will be investigated. In this case a translator will be implemented to generate the dbl configuration files from the XML definitions. The intention would be that a central XML configuration file would be used to generate all dbl files. In this fashion the TICS configuration file management would reflect, and test, the final XML based configuration database.

8.2 Run Time access to configuration data

At run time there will be a Configuration Database process (called `Environment`) running on each host (Workstation and LCU) and containing the branch of the configuration hierarchy pertinent to that host.

It is not possible to modify dynamically at run time the structure of the configuration database, but it is necessary to stop, regenerate and restart the Environment. It is possible to modify dynamically at run time all values stored in the Configuration Database.

C++ applications can access (read/write) the configuration database using an access API.

Other CORBA applications will use a CORBA based Database Service through and access IDL, that will provide the same access capabilities in a language independent way but with lower performance.

The interface, provided by ACS 1.0, is conceptually of the type:

- value db.get(name)
- db.set(name, value).

Value may be any of the allowed primitive types (integer, float, string and array of Bytes….).

The hierarchical nature of the database is defined simply by providing a hierarchy in the name string. Here db is a reference to a database object that has been previously opened.

8.3 Warm Start

A particular requirement for TICS is for “warm starts.”

It must be possible to restart a subsystem merging:

- Default configuration values
- User specified values
- The latest "current" value

The ACS 1.0 Configuration Database provides the basic building blocks to implement these features:

- The dbl configuration files provide the default configuration. At any time it is possible to stop-restart the configuration database with the pure set of default values (eventually after having edited the dbl configuration files).

- At any time it is possible to create a snapshot of the current complete Configuration Database running on a host (dbForseSnap command). This snapshot can be used at restart time instead of the dbl configuration files.
- At any time it is possible to backup and restore branches of the configuration database (dbBackup and dbRestore commands). The commands can be used to load user-defined values at any time or to save preference values after having modified the values at run time.

9 Human Interfaces

9.1 Scripting

G. Harris
Last Changed: 2001-02-06

The scripting language for the Test Interferometer is Python. Its most important characteristics for our use are that it:

1. is completely object based,
2. has cleanliness and rigor in its grammar,
3. supports access to all ALMA components via direct calls or CORBA.

We will also need to transform between GUIs, XML, and executable procedures to keep ALMA simple and stable. The ability to make mathematically rigorous transforms between various procedural forms requires a simple, preferably left corner, grammar. Python supplies that grammatical rigor[3].

Python provides other attributes desired for the test interferometer, including:

1. CORBA support, using standard ORBs
2. XML support, including SAX and DOM parsers.
3. Database access to RDBMS and object based storage, via ODBC, JDBC and persistence mechanisms.
4. Graphical support from either Tk or Java.
5. Interactive source code debugger.
6. Extensive libraries from the scientific community including astronomy.

[3] A left corner grammar is just a left sided grammar for which a recursive descent parser is not necessary.

Python supplies standard object oriented operations of classes, inheritance, polymorphism, etc. Since it is totally object based, it provides complete run time dynamic object creation, assignment, typing and parameter checking. It also provides file and structure conventions for development including classes, namespaces and functions in packages and modules.

Python is available in two forms:

1. Python - a bytecode C based compiled interpreter with C++ and Tk libraries [the most mature, heavily maintained and fastest], and

2. Jython - a JVM bytecode based version which can access Java and its libraries.

3. Python can also be converted to C and compiled; however it is not clear how well supported this application is.

9.2 Source Catalogs

R.W. Heald

Last Changed: 2000-10-21

A catalog is used to select a celestial object and provide the parameters needed to point the ALMA antennas to it. The user is given a "source browser" GUI containing a display of the list of objects that meet a given criterion, by source name, or by coordinates. The user is then able to make a selection from the list. This capability is useful in both observing preparation and during on-line observing, although for the TI only the latter will be used.

There will be several catalogs derived from different resources, and users will select a particular catalog before viewing the objects in that catalog. All catalogs are one of two types, either for "stationary" objects, or for moving objects such as comets, asteroids, planets, and satellites.

For stationary objects, each catalog entry is one line long. Each catalog entry has a format as below (Headings are not part of the catalog; Sample data is shown).

Name	System	X Coordinate	Y Coordinate	X PM	Y PM
3C284	J2000	13h11m04.7s	+27d28m08s	0.0	0.0

Frame	Type	Velocity	f1 Flux	f2 Flux	f3 Flux
HEL	RAD	71770	0.0	0.0	0.0

- The system can be B1950, J2000, APPARENT, ECLIPTIC, GALACTIC, or HORIZON.
- Depending on the system the X and Y coordinates are either equatorial right ascension and declination, ecliptic longitude and latitude, galactic longitude and latitude, or horizon elevation and azimuth.
- Proper motion for each coordinate in milli-arc-seconds per year.

- The source velocity reference frame can be either

  LSR (local standard of rest velocity reference frame),

  HEL (heliocentric velocity reference frame),

  GEO (geocentric velocity reference frame), or

  TOP (topocentric velocity reference frame).
- The source velocity type can be either

  RAD (radial velocity type),

  OPT (optical velocity type), or

  REL (relativistic velocity type).
- The source velocity in kilometers per second.
- Flux densities in milli-Janskys for the three Test Interferometer frequency bands (35, 85, and 225 GHz).

The VLA calibrator source list will be provided as a basic stationary object catalog.

The catalog for a moving object requires an ephemeris for each object. An ephemeris is a table of positions where each entry gives the object's position for a particular time. The time interval between entries is constant. The object's current position is found using interpolation, thus, the total time interval given by the ephemeris must include the time at which the object is to be tracked.

An ephemeris begins with a header containing the object's name, and the object's velocity reference frame and type given in the ephemeris. The choices for velocity frame and type are the same as those given for the stationary object catalog entries above. An example header might look like:

```
OBJNAME Hyakutake
VELFRAME LSR
VELTYPE RAD
```

Each ephemeris entry has a format similar to below (Headings are not part of the ephemeris; Sample data is shown). This format allows direct input from the JPL Horizons system. See http://ssd.jpl.nasa.gov/horizons.html.

Date/Time	Right Ascension	Declination	Distance	Velocity
2451675.458333	04 11 54.0991	+21 36 17.384	2.488596948	5.99239

- TT (Terrestrial Time) is given in either a string (various formats) or Julian Date with fractional day (shown). Notice TT used to be called Terrestrial Dynamical Time, or TDT. TT uses ordinary SI seconds, and it is tied to TAI through the formula TT = TAI + 32.184s.

- Position is given in geocentric J2000 astrometric right ascension (in hours, minutes, and seconds) and declination (in degrees, arc-minutes, and arc-seconds).
- Geocentric distance is given in Astronomical Units (AU).
- Geocentric velocity is given in kilometers per second.

Software will be provided for generating the current ephemerides for most planets, the moon and sun.

For both catalog types, the software handles manual input or from an ASCII file. A large variety of input formats for coordinates and date/time are accommodated. Coordinate formats include time and degrees. Date/time format includes the familiar year, month, day, etc., and Julian and Modified Julian Date. The main restriction occurs in the order of the fields.

Objects in the catalog can have alternate names, known as an "alias". The catalog can add an object's alias, and can find an object by any of its names. This is implemented as an additional table indexed by the object's name with a pointer to the catalog and entry where the object's entry exists.

The source catalogs interface to the rest of the system in a couple of ways. First, an observing script is able to directly access the catalogs using a programmatic interface. This interface has the following functions:

- `openCatalog(catalogName)` - This function returns a pointer to the catalog and the catalog type, or a `NULL` if the catalog is not found.
- `closeCatalog(catalogPointer)` - This function terminates access to the catalog.
- `getCatalogEntry(catalogPointer,objectName)` - This function returns the entry, or a `NULL` if the entry is not found.

A second catalog interface allows the catalog GUI to command the antenna(s) to track an object selected from a catalog. To accomplish this the GUI will directly command the antenna mount device.

A software package that provides many of the desired features for the ALMA source catalogs is called SkyCat, and is available from the VLT software group. Their latest version is a set of JavaBeans called JSky. It needs further investigation as to its suitability. See http://archive.eso.org/Jsky.

9.3 GUIs

F. Stauffer
Last Changed: 2001-11-16

GUIs are for TICS are JAVA or Python (using TkInter) based, and for ACS are JAVA or Tcl based.

Antenna GUI

- Source trajectory
- Command position
- Current position
- Cable wrap

Logging GUIs provided by ACS

- Errors, alarms, other logging
- Debug
- Filter on severity and type and time
- Scrollable buffer
- Selectable time range

Scripting command interface

- List scripts
- Execute script
- Create/edit script
- Command line interface

Monitor and control

- Parameters, configurations, monitor, and control points displayed and edited using a hierarchical decomposition.
- Multi-channel chart recorder.
- Type in value.
- Archived monitor point display.

Engineering displays

- Front Ends
- Correlator
- Round-trip phase correction.
- Tracking/drive system.
- Downconverter
- Secondary
- Nutator

Other GUIs

- On-line data display (quasi-raw backend values)
- Doppler tracking
- On-line catalog browser

9.4 Engineering Interfaces

M. Brooks
Last Changed: 2001-05-05

In general, device designers will develop test and monitoring software for use while the hardware is still "on the bench". It is a goal of the TICS that such software should be available when the device is installed in the field for hardware diagnosis and debugging. Since most hardware sub-systems will have a CAN interface, it is always possible for an engineer to directly connect to the device interface after removing it from the antenna CAN bus. This allows the use of any test code, but requires a physical presence at the hardware itself. This approach allows an independent means of testing hardware without using the TICS software.

If a device designer uses the LabView system to develop bench test software, the TICS may (depending on Engineering interest) provide a method of accessing the device over the network from a location removed from the actual hardware. This would require a substitution of the CAN access LabView routines with software to access the same monitor and control points through the TICS.

In addition, ACS will provide the ability to build simple GUI interfaces which can get and set monitor and control points through the TICS. ACS will also provide a standalone tool to get/set Monitor points – the Object Explorer. In general, no calibration or analysis capabilities would be provided for such an application.

10 Time

10.1 Synchronization

F. Stauffer
Last Changed: 2000-02-07

ALMA's control system achieves real-time behavior in a large non real-time networked system by distributing a timing event and time tagging monitor and control as needed. ALMA's master clock counts timing events.

Usually, the user executes commands on the ACC. The commands are either synchronized to the timing events or not, with the majority of commands not synchronized. Synchronized commands are tagged with the array time of the timing event to start on, which is at least the current time plus an estimate for the latency[4]. Latency is the time for the command to setup the hardware, plus network and OS indeterminacy.

The CAN hardware timing specification is given in Brooks and D'Addario (2001). Most devices do not have precise timing requirements. For those that do, the control software must arrange for control commands to be sent to the device within the first 24 milliseconds after a *timing event*. Monitor requests must be sent within a window beginning 24 milliseconds after a *timing event*, and ending 4 milliseconds before the next *timing event*.

To allow the time critical commands and monitor requests to meet this constraint the time system provides semaphores that release tasks at the *timing event* and/or 24 milliseconds after a *timing event*. The real-time computer on-board timers are used to provide this service.

Test correlator timing is specified by Pisano (2001). The only timing synchronized command is the start integration. In addition there is a 'synchronized to timing event' mode which when set means dumps occur on the even timing event, otherwise the dumps are free running after

[4] Although probably considerably longer than necessary, we will probably start by using 1s as this value has been used successfully at other telescopes.

the initial start timing event. The time to start the integration requires the command to be sent from the CCC to the correlator hardware 72 to 96 ms before the start time (not including the network and OS indeterminacy). The stop integration needs to be at least 72 ms before the end of integration (or alternatively it can run for a set number of integrations).

The time synchronized devices are:

- correlator
- phase switching and fringe rotation
- nutator
- antenna motion
- total power
- holography

Commands are passed from ACC to the real-time computers over the network using CORBA. In the ALMA Common Software Technical Requirements, Raffi and Glendenning (2000) specify a minimum of 1000 messages per second. The messages are a limited resource, so a strategy to improve performance may need to be used.

If performance is a factor, then the following can be implemented:

- Download configurations (observing tables) to the real-time computers.
- High level commands will replace common command sequences on the ACC, and the command sequences are moved to the real-time computers.

Data produced is time tagged. Data that requires quality checks needs to be buffered long enough to accommodate network and OS indeterminacy. This is discussed in the "Data Production" section later.

10.2 Master Clock

F. Stauffer
Last Changed: 2001-10-30

10.2.1 Description

The ALMA Master Clock is implemented on the central real-time computer, ARTM, because it is responsible for the Central Reference Generator (CRG) timing hardware, Figure 1. The CRG generates a timing event to synchronize hardware and software, and it generates the reference frequencies used by the hardware. An external precision oscillator is the reference frequency for the CRG. Hardware events can be aligned to the oscillator clock and have an ambiguity of the timing event. The Master Clock resolves the timing event ambiguity. D'Addario (2000) describes the timing system.

The CRG is used to align the ALMA clock to an external 1-PPS and to monitor the difference between external time and the ALMA clock. An alarm is generated when the difference between the ALMA clock and the external clock is outside of a settable range. External time is provided by a GPS receiver attached to the ARTM. The 1-PPS is aligned to TAI (actually GPS time, however GPS time is a constant offset from TAI), and hence the whole time system is aligned to TAI on reset, although it will drift very slowly thereafter.

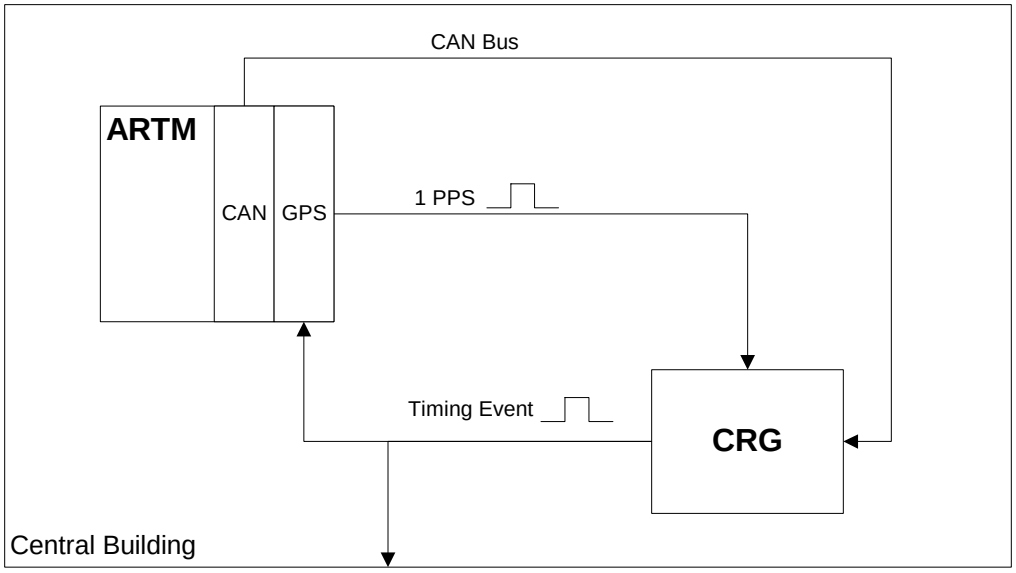


Figure 6 - Computer and electronics hardware needed for the master clock. Hardware for distribution, oscillator, etc. is not shown.

10.2.2 Parameters

- TAI when array time synchronized to 1PPS
- TAI of last timing event
- Last timing event count - internal count zeroed when CRG synchronized to 1PPS
- Drift between array time and TAI: array time - TAI

10.2.3 Commands

- Synchronize - synchronize CRG to TAI with 1 PPS (executed very infrequently)

10.2.4 States

The ENABLE state, Figure 2, has the following machine with these states:

- IDLE
- SYNCHRONIZING - synchronizing CRG to TAI with 1 PPS

- CLOCKRUNNING - track array time - TAI drift, count timing events

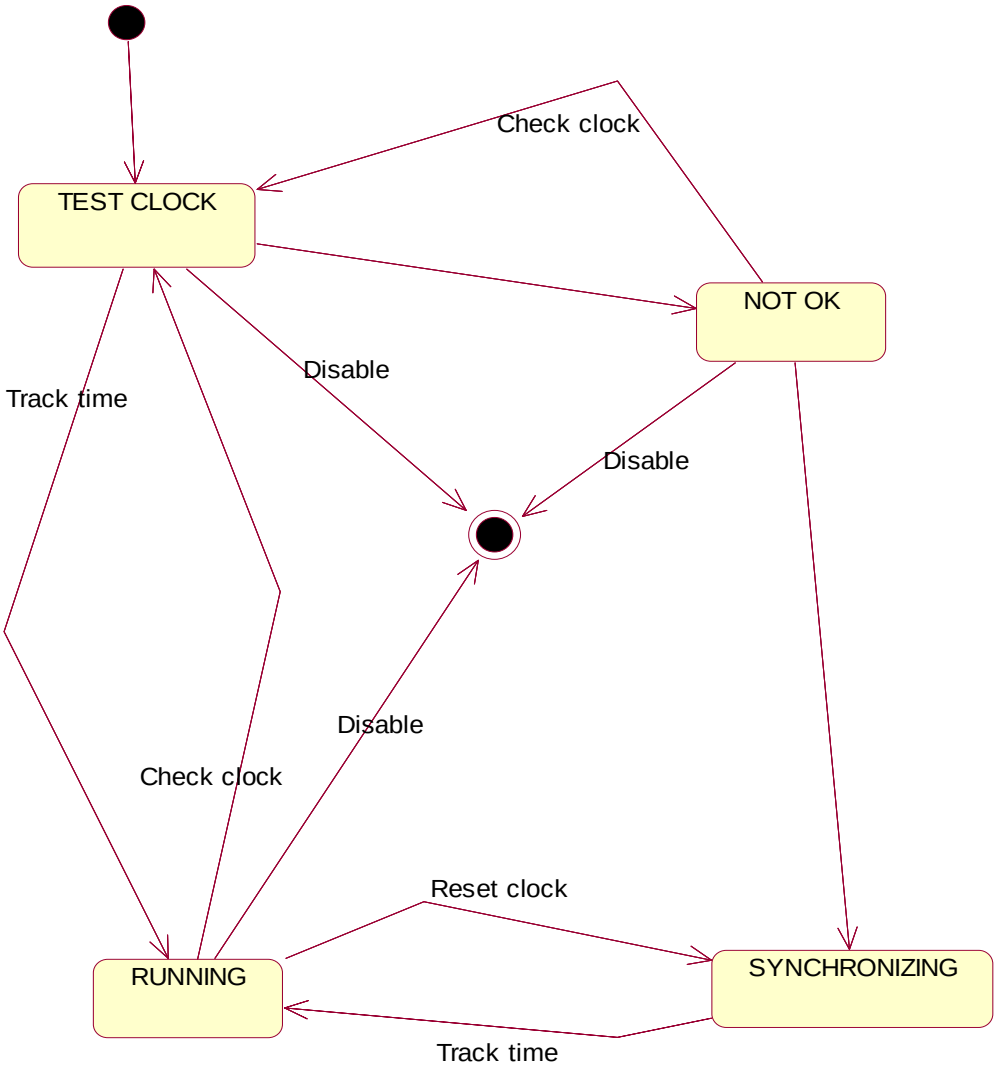


Figure 7 - Master clock state machine (sub-states of ENABLE).

10.2.5 Timing

Figure 3 is a timing diagram that shows the relationship between the GPS 1-PPS and the CRG counter reset. The reset counter CAN command is sent one or more timing events ahead of the desired 1-PPS.

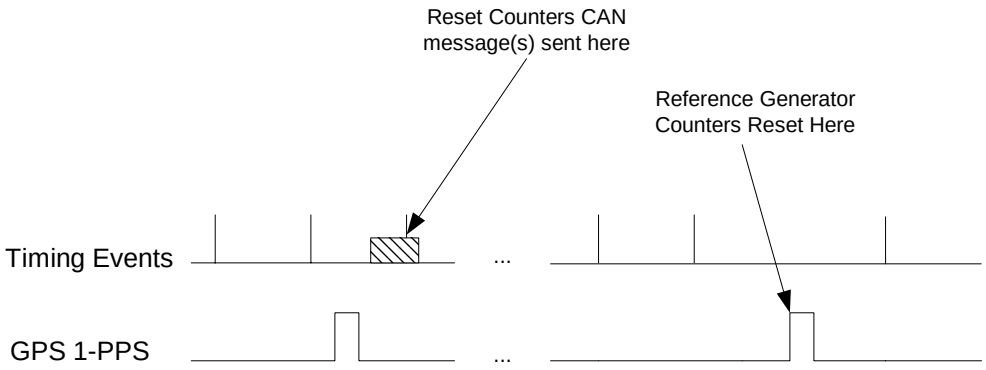


Figure 3 - Timing diagram showing the relationship between the GPS 1-PPS and the CRG counter reset.

10.2.6 Class Diagram

Figure 4 is the Master Clock class diagram.

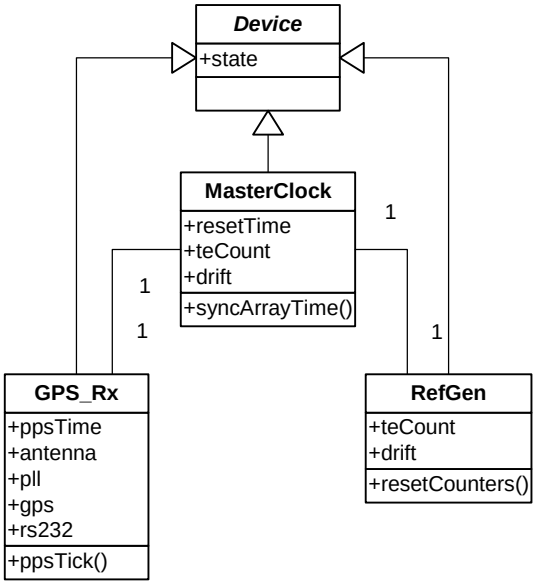


Figure 4 - Master Clock class diagram.

10.3 Time System and Representation

Ron Heald
Last Changed: 2001-04-10

The fundamental ALMA time system is International Atomic Time (TAI). TAI is based on the SI second, and is continuously increasing, with no discontinuities, i.e. no leap seconds. It was chosen primarily for these reasons.

ALMA will use *array time* for all internal time. *Array time* is, for most purposes, identical to TAI. *Array time* is maintained by the central Master Clock, and is initially set to TAI using a Global Positioning System (GPS) receiver. Thereafter, *array time* is driven from a local maser clock, and therefore it may slowly drift from TAI. Because of the possible drift, *array time* cannot truly be called TAI. See the section "Master Clock" for more information.

Although TAI is the system used by the software, other time systems such as UTC and local sidereal time are displayed and accepted in user interfaces. In particular, UTC, for the most part, is the only time seen by the user. These other time systems are produced by converting *array time*, and as such, contain the same drift from their "real" value as *array time*.

A central database of leap seconds is maintained. The database contents allow TAI to be easily converted to UTC, and vice versa. Notice it does not particularly matter if there is a discontinuity in UTC during a leap second because it only affects the displayed time. The database contents are based on information from the International Earth Rotation Service (IERS); See http://hpiers.obspm.fr.

An unsigned 64-bit integer (C-language "long long int") is used to encode a particular time. The integer gives the number of 100 nanoseconds that have passed since 15 October 1582 at 00:00, which most people recognize as the start of the Gregorian calendar. This encoding gives a range from the start time until March 11, 60038. This is the same encoding used in the CORBA and X/Open DCE Time Services, and by the Posix.4 standard.

A period of time is encoded using a signed 64-bit integer giving the number of 100 nanoseconds, similar to the time representation. This encoding gives a range for durations of zero to more that ± 29,227 years.

It is recognized that these representations are quite detailed, and provide more accuracy than any current computer can act upon. However, the ALMA calculated results are needed at 100-nanosecond resolution (and better), and this is a reason the representation was chosen. It is also thought that the excess capability may prevent having to change the representation in the future.

Two classes, "Epoch" and "Duration" are provided for managing particular times and durations, respectively. Objects created from these classes hold a particular time or duration, and have methods for operating on their value.

Additional class methods are provided to set and extract the internal 64-bit integer representation. The purpose of these methods is to allow the sending of time and duration data across the network while avoiding remote object invocation and its inherent inefficiency. The idea is to always have local Epoch and Duration objects and to only pass the time datum between machines.

CORBA's objects-by-value specification may be a better way of dealing with this problem. However, this specification was only released with CORBA 2.3 and it not yet widely supported. It is being further investigated.

In the following, let "EpochS" be the 64-bit integer representation of an *array time*, and "DurationS" be the 64-bit representation of a duration. The Epoch class has the following attributes:

- Has attributes for integer *array time* piece parts (year, month, day, day of year, hour, minute, second, and microsecond) to set and retrieve the object's value.
- Has an attribute for an EpochS to set and retrieve the object's value.
- Has a method to accept UTC or TAI time in string form to set the object's value.
- Has a method to accept a format string and return the object's value as an *array time*, UTC or TAO time in string form.
- Has a method to compare this object's value with another Epoch object's value and return either "equal", "less than", or "greater than".
- Has a method to compute the difference between this object's value with another Epoch object's value and return a Duration object containing the result.
- Has methods for adding a Duration object's value to this object's value, and for subtracting a Duration object's value from this object's value.

The Duration class has the following attributes:

- Has attributes for integer piece parts (day, hour, minute, second, and microsecond) to set and retrieve the object's value.
- Has an attribute for a DurationS to set and retrieve the object's value.
- Has a method to compare this object's value with another Duration object's value and return either "equal", "less than", or "greater than".
- Has methods for adding another Duration object's value to this object's value, and for subtracting another Duration object's value from this object's value.
- Has methods for multiplying and dividing this object's value by an integer.

10.4 Time Distribution

R.W. Heald
Last Changed: 2001-04-12

To achieve synchronization across the array it is necessary to distribute timing information. The information has two forms: as the 48-millisecond *timing event* distributed by the hardware and as a master clock distributed by the software.

It is a principle of the design that timing information is obtained only from a single source, and that a computer shall never exchange timing information with others outside of the

source. The Central Reference Generator (CRG) is the creator of *timing events* and their only source. The central Array Real-Time Machine (ARTM) is the keeper of the master clock, and is the only source of *array time*.

The master clock time is initialized to TAI using the output from a GPS receiver. This initialization is done only rarely, and thereafter *array time* and TAI may slowly drift apart. Otherwise, *array time* is not distinguishable from TAI. See the section "Master Clock" for more information.

Timing Event Delay

The maximum propagation delay of the *timing event* from the CRG to the furthest antenna (~25km) is about 170 microseconds. This delay is ignored in the setting and maintenance of clocks.

However, the propagation delay must be accounted for in the fringe rotation timing and the synchronizing of phase switching with the correlator. For these purposes, a table of the actual delay to each antenna is maintained in the Array Control Computer (ACC), and made available to other systems that need it. The table is based on measurements and is accurate to about 1 microsecond. The table only needs updating when an antenna is moved or hardware is replaced (D'Addario, 2000).

Master Clock Time Server

Two servers are implemented on the ARTM. First, the ARTM acts as a Network Time Protocol (NTP) server providing *array time* to the central workstations, like ACC. These machines are not real-time and do not require precise time. This time is used, for example, to time-stamp the machine's local files.

A second server protocol is implemented on the ARTM to provide the master clock to clients that need it. These clients will be only real-time computer systems. When requested the server provides the *array time* of the current *timing event*. The server waits until just after a *timing event* before sending its response. This allows the response to have nearly the full 48-milliseconds to reach its destination. To guarantee there is no ambiguity about to which *timing event* the response corresponds, the client measures the period between sending its request and when the response is received. The measurement is done using a local on-board timer. If this round-trip time is too long, the process is repeated until the round-trip time is within the proper range.

LCU Clock Requirements

The ALMA real-time computers require precise time (considerably better than a *timing event*), and therefore maintain a local clock that is "slaved" to the master clock. These computers are sometimes called Local Control Units (LCU), and so the clock is known as the "LCU Clock".

One such computer is the Antenna Bus Master (ABM). There is an identical copy of the ABM located at each antenna. Among its duties, the ABM performs the equatorial to horizon coordinate transformation and requires accurate time for this reason. The ABM must also

accurately time-stamp the total power, holography, and OTF position data that it buffers and sends to the center.

The Correlator Control Computer (CCC) is located at the correlator, and also requires precise time, as it must accurately time stamp the data it produces and sends to the archive.

The ARTM also needs precise time. Its needed to control several devices located at the array center that are part of the ARTM's responsibilities.

Certain commands and monitor requests are associated with *timing event*s and their execution timing is critical (Brooks and D'Addario 2001). These are known as "critical" commands and monitor requests. To enable applications to meet the timing requirement certain synchronization methods are provided. The methods provide for an application wanting to send a time-critical command to be awakened when the *timing event* occurs. For applications wanting to send a time-critical monitor request, the methods awaken the application 24 milliseconds after the *timing event.* One way this can be accomplished is for the LCU clocks to directly provide such methods.

Another way to accomplish this synchronization is to use the methods provided by the ACE Reactor. The Adaptive Communications Environment (ACE) is required by TAO CORBA, and will be part of the system in any case. The Reactor uses the underlying clock, and provides for timeout events and handlers that can be used to achieve the necessary timeout events.

The purpose of the LCU clock, then, is to provide the real-time systems with accurate and precise time, and with methods for synchronizing commands and monitor requests. The required resolution of the LCU clock is about one millisecond. This requirement primarily comes from the need to supply precise time tags for collected monitor data.

LCU Clock Initialization and Maintenance

The LCU clock sets its time by obtaining the *array time* corresponding to a particular *timing event* from the master clock. This is done using the master clock server described previously. The master clock server is only used during clock initialization and when a reset is required. For testing, the clock can be set using a user-supplied time instead of using the master clock server.

Once a clock has been initialized, it independently maintains the time thereafter by counting *timing events*. All LCUs directly receive the *timing event* signal. The signal is connected to each LCU such that a processor interrupt is generated whenever a *timing event* occurs. One of the LCU clock's primary responsibilities is to receive and process each *timing event* interrupt.

A local watchdog timer is used to detect if any *timing event* is missed. If this should occur, the clock enters a fault state. From the fault state, the clock attempts to re-synchronize using the master clock server. During this period the clock continues to function normally as there are activities, such as monitor point logging, that require time to be available, but do not need the synchronization provided by the *timing events*

VxWorks Clock

The VxWorks system clock is set for 125 ticks per second. This makes the *timing event* period evenly divisible by the system clock, i.e. there are nominally six VxWorks' clock ticks for every *timing event*.

ALMA has chosen the MVME2700 PowerPC (PPC) Single Board Computer (SBC) for their real-time systems. The VxWorks operating system runs on these systems. To implement its system clock, VxWorks uses a decrementing counter that is part of the PPC604 processor chip. An oscillator, that is also on-board the chip, supplies the input clock to the counter. The oscillator has an operating frequency of 16,666,666 Hz (60 nanosecond per tick).

To generate the system clock interrupt, the decrementing counter is initialized for 133,333 ticks (1/125 second). When the counter reaches zero a system clock tick is generated. This process is then repeated for each system clock tick. For more information, see the VxWorks' source file ppcDecTimer.c.

By reading the decrementing counter on the fly and adding the result to the count of system clock ticks, times with the resolution of the decrementing counter are obtained. VxWorks uses the decrementing counter in this way to provide the "timestamp" capability, and one can obtain the same results by calling the system function "sysTimestampLock".

The *timing event*s are used to synchronize the system clock with the master clock. To accomplish this the *timing event* interrupt service routine cause the decrementing counter to be reset to its initial value. Using this technique, the system clock is able to run with or without the *timing event* interrupt. When *timing event*s are not available the system clock is not synchronized to the master clock, but is otherwise fully functional. For testing, the *timing event* signal, even though it may properly be creating interrupts, can be ignored by changing a software switch.

Notice the system clock will run slightly behind the *timing event* by the interrupt latency time. This time has been measured at around 5 microseconds, and is not thought to be significant.

Clock Class

This class provides functionality on both LCUs and general-purpose workstations. It is provided in the form of a CORBA interface making it available for use by all programming languages. Using this class, an application can obtain the current time and register for timeout events. For efficiency, it will always be used locally.

To implement the timeout events the class uses the ACE Reactor mechanism. ACE will be present on both LCUs and workstations since its required by the TAO CORBA. Through the CORBA interface a timeout event can be set for a single specified time, or it can be set to occur periodically with a given uniform spacing. It can also be set to begin at a specified time and occur periodically thereafter at a given rate. Event status can be checked at anytime, and events can be canceled at anytime before they occur.

11 Devices

11.1 Control Style

Control of the system is hierarchical, or master-slave. The top level control process controls high-level software Device abstractions, which in turn control lower level Devices until we get down to individual Device controllers, which are our lowest level interface in the system and correspond to real hardware. For example, a high-level Mount device would be commanded in RA and DEC. In turn, it would control the actual Antenna Control Unit (ACU) in azimuth and elevation.

A consequence of this choice is that devices cannot "pull" required information themselves from its ultimate source. For example, part of the conversion of RA/DEC to AZ/EL requires some weather parameters (for refraction calculations). Following the master-slave control philosophy, the weather parameters are passed to the high-level mount Device rather than having it retrieve the values itself directly from the weather station or monitor point archive.

11.2 CAN Interface

M. Brooks
Last Changed: 2001-11-01

The CAN bus has been selected as the primary interface to distributed hardware devices within the ALMA project. CAN is an ISO standard multi-drop communications medium used extensively in automotive and industrial applications. A higher layer master/slave protocol has been defined in Brooks and D'Addario (2001) which governs the behavior of slave nodes at the devices. This bus is hereafter referred to as the ALMA Monitor and Control Bus (AMB). That memo also defines the timing specifications for communicating with devices which have synchronization associated with 48 ms Timing Events. A distributed reset pulse is also defined for the purpose of remotely resetting all nodes on a CAN bus.

In the Test Interferometer there will be a number of different operational CAN buses:

- At least one at each antenna for M&C (most likely two)
- One at each antenna for total power samples
- One in the central control building for M&C

11.2.1 Bus Master Nodes

It is required that on any single AMB there be only a single bus master node. Bus master nodes have been coded and tested for the following hardware and software combinations:

- O/S: VxWorks. SBC: MVME2700, MVME1603, MVME2604. CAN Interface: Tews Datentechnik TP816 Dual and Single Port PMC CAN modules (available in the U.S. from SBS Greenspring)
- O/S: Windows NT. Any PC platform with PCI. CAN Interface: National Instruments Dual Port PCI CAN board. LabView bus master routines.

It is the responsibility of the ALMA Computing Group to design, implement and test all code developed for these bus master platforms.

Figure 8 illustrates the class diagram for the CAN access package available for the VxWorks platform. The classes shown represent the client interface to the AMB, which is essentially a polled resource. Clients create objects representing physical CAN nodes (ambSimpleCANNode or ambCANNode classes) and may call read or write methods to effect monitor and control transactions to the physical slave node. The ambCANManager class is a singleton pattern which manages all of the physical CAN ports of the ABM or ARTM. Each port has a dedicated server process which services priority queued requests and keeps local performance statistics (not shown in the diagram). The manager object provides a method for finding devices connected to the bus by the device serial number.

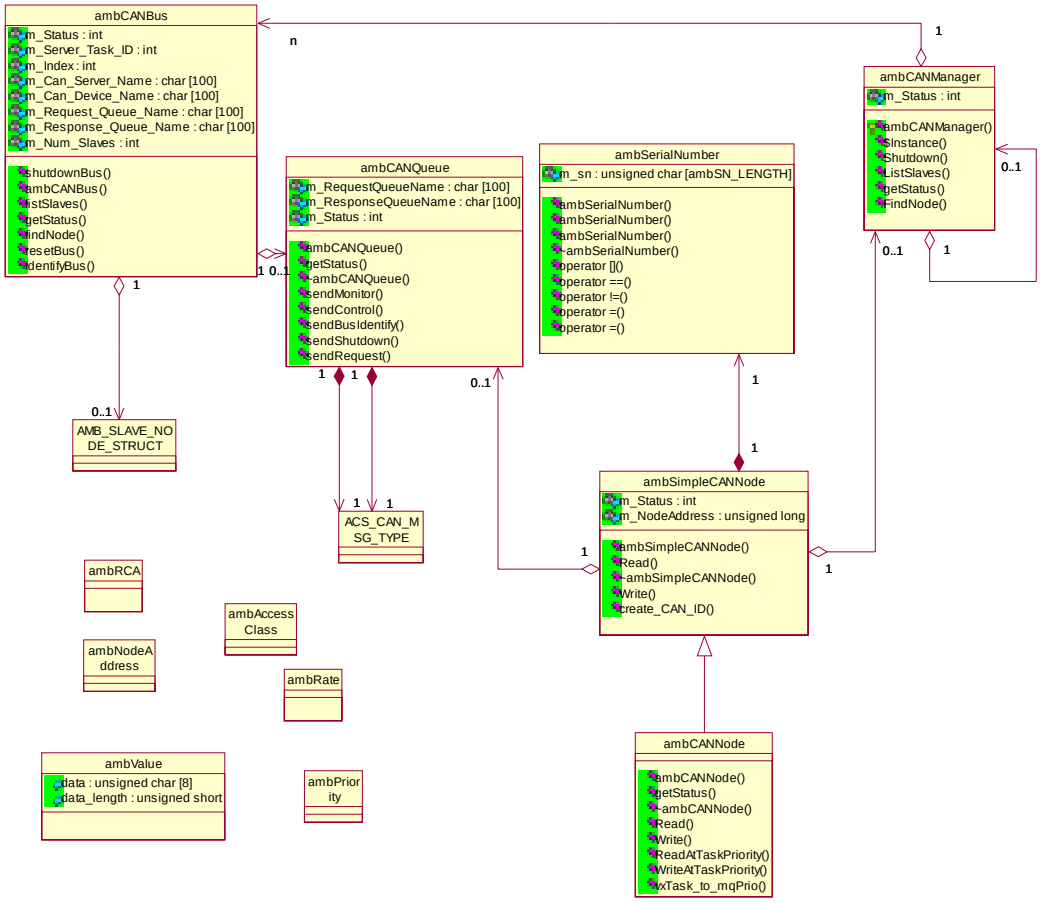


Figure 8: Class Diagram for VxWorks CAN access package

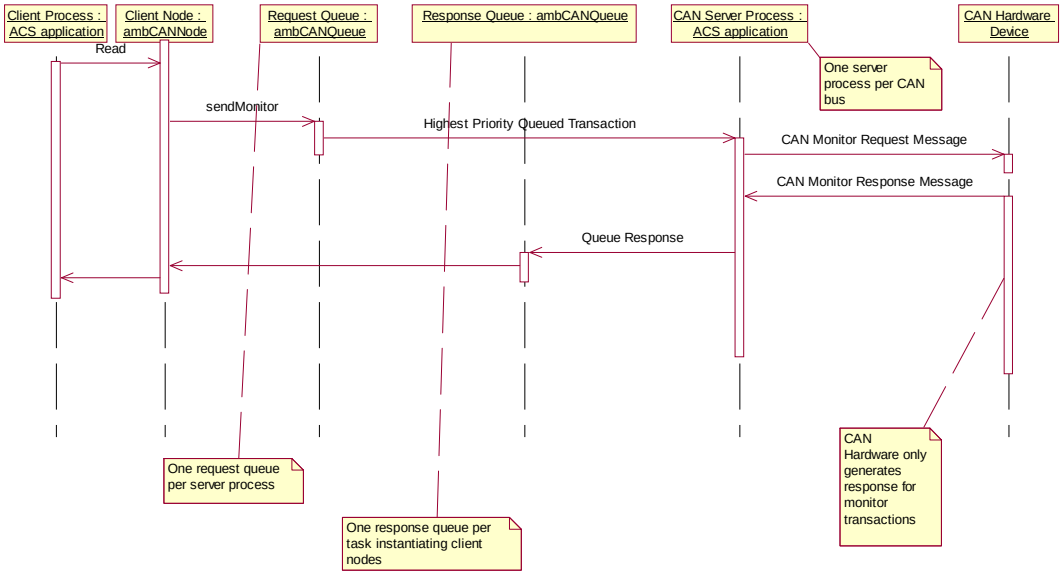


Figure 9: Sequence Diagram for Monitor Transaction within VxWorks CAN access package

11.2.2 Bus Slave Nodes

There are currently two designs for circuits allowing the connection of hardware devices to the ALMA Monitor and Control Bus (AMB) in response to the requirements defined in Brooks and D'Addario (2001). These two circuits are referred to as the AMB Standard Interface, Type 1 (AMBSI1) and the AMB Standard Interface, Type 2 (AMBSI2). All designers of hardware interfacing to the AMB should use one of these two; new slave node implementations may be considered where sufficient justification can be provided. All slave node implementations must comply with the bus specification detailed in which requires that no slave node initiate transactions on the bus unless polled by a bus master.

The AMBSI1 and AMBSI2 have been developed to serve the needs of device designers requiring varying degrees of complexity in local hardware monitoring and control, and for accommodating different physical size constraints. The AMBSI1 requires space for a Eurocard 3U height board and associated connectors. It provides a powerful 16-bit micro-controller with default firmware providing a small set of I/O capabilities. Device designers may also implement additional device-specific code to access the serial ports or to interface to devices on the external bus. The use of device-specific firmware is expected to be the main area of deployment for the AMBSI1.

Thus, the purpose of the AMBSI1 is to incorporate the AMB protocol and device-specific functionality onto the one board. Conversely, the AMBSI2 is a very small daughter-card for use in devices which either have their own microprocessors or need a very minimum amount of I/O to the CAN bus. There is no provision for device designers to add device-specific firmware to the AMBSI2.

The following additional slave node CAN interfaces will be supported in addition to the AMBSI2 and AMBSI2:

- O/S: Linux. Any PC platform with PCI. CAN interface: Janz CAN-PCI/K2O. mainly for use in slave node simulation in the lab.
- CompactPCI CAN board: planned for use by Vertex ACU, based on the Philips SJA1000 CAN controller ship.
- O/S: Windows NT. Any PC platform with PCI. CAN Interface: National Instruments Dual Port PCI CAN board. LabView bus slave routines. For use in simulating slave nodes in the lab.\

11.2.2.1 AMBSI1

The purpose of the AMBSI1 is to provide a highly flexible interface board with on-board CAN and device-side I/O consisting of parallel, serial and bit-wise ports. The board would be delivered to hardware designers with a standard firmware package supporting basic CAN access to the I/O ports and external bus. The micro-controller on the AMBSI will have sufficient spare processing capacity to run additional user-specific code, such as closing fast sub-millisecond control loops. The board also provides for the Global Slave Reset pulse and 48 ms Timing pulses to generate appropriate signals to the micro-controller.

The ALMA Computing Group is responsible for developing any device-specific code in conjunction with the hardware developer and for integration testing the product to ensure that the user code does not interfere with the CAN slave code. All AMBSI boards should be similar in hardware, and identical in size and connector arrangement. Code may be loaded into on-board flash memory by means of the CAN bus or the local RS232 port.

The CAN bus servicing code is written so as to be entirely interrupt-driven. It runs only in response to interrupts from the CAN controller, and those interrupts are set to the highest possible priority (in the Infineon C167, this is interrupt level 15, group level 3). When not servicing an interrupt, the processor executes an idle loop. User-written code may include a function to execute from the main idle loop, or it may simply consist of the callbacks for CAN message reception interrupts. Self-test routines should also be included in user code.

11.2.2.2 AMBSI2

This device would perform the function of a CAN bus to Serial Peripheral Interface (SPI) converter. It is designed for use in systems which either have their own microprocessors or need a very minimum amount of I/O to the CAN bus. It is currently proposed that the resulting subsystem would require +5 Volts at less than 2 ma and occupy a small daughter PCB of approximately 2 in by 1 in. The PCB would be either mounted to the main PB Board by several multi-pin headers which also serve as the I/O connectors for the subsystem.

There would be about 16 bytes of RAM which could be accessed by either the CAN bus or the SPI port. An "Attention" output would indicate when RAM contents had been altered by the CAN bus. The unit will have its own oscillator and reset circuitry, completely independent of the host microprocessor. All of the software in the subsystem would be developed by and be the responsibility of the software group.

In addition to providing the CAN interface, the unit would also provide the interface for both the Global Slave Reset and 48 ms Timing pulses which come in through the AMB DB-9 connector or module wiring as RS-485 and be available as TTL signals to the application circuitry.

11.2.2.3 CAN Device Initialization in TICS

The AMB protocol defined by ALMA is capable of run-time discovery of devices. In response to a bus initialization request all nodes on the bus will respond to that request by returning their node and serial numbers. This facility is intended to allow the dynamic discovery of which devices are available in the system.

All devices that might be attached to the system have an entry in the configuration database containing their configuration parameters. In addition the database has a section that returns the name of the Device given its serial number. This name is the same as the name of the device in the configuration database.

Initialization is driven by a startup script that operates as follows:

1. The CAN access package is used to issue a bus initialization request (at least this method is assumed to be bound to IDL). The returned node and serial numbers are captured.
2. The serial number is used to lookup the device name in the database.
3. The serial number and node number for the device with that name will be inserted in the configuration data for that Device by the startup script. In particular, the node number will be needed so that the *CAN Properties* can be constructed properly. Any other dependent configuration values can also be modified at this time.
4. Available devices are also written into the Activator database record so that the Manager knows they are available to be started.

The only missing facility presently from this procedure would be the ability to instantiate devices of a known type whose actual existence in the system has not been foreseen (that is, a database entry for that name has not yet been written and inserted into the running system). This is due to the fact that the "structure" of the current ACS configuration database may not be changed at runtime. This restriction does not appear to be serious.

11.3 CAN Device Properties

M. Pokorny
Last Changed: 2001-05-08

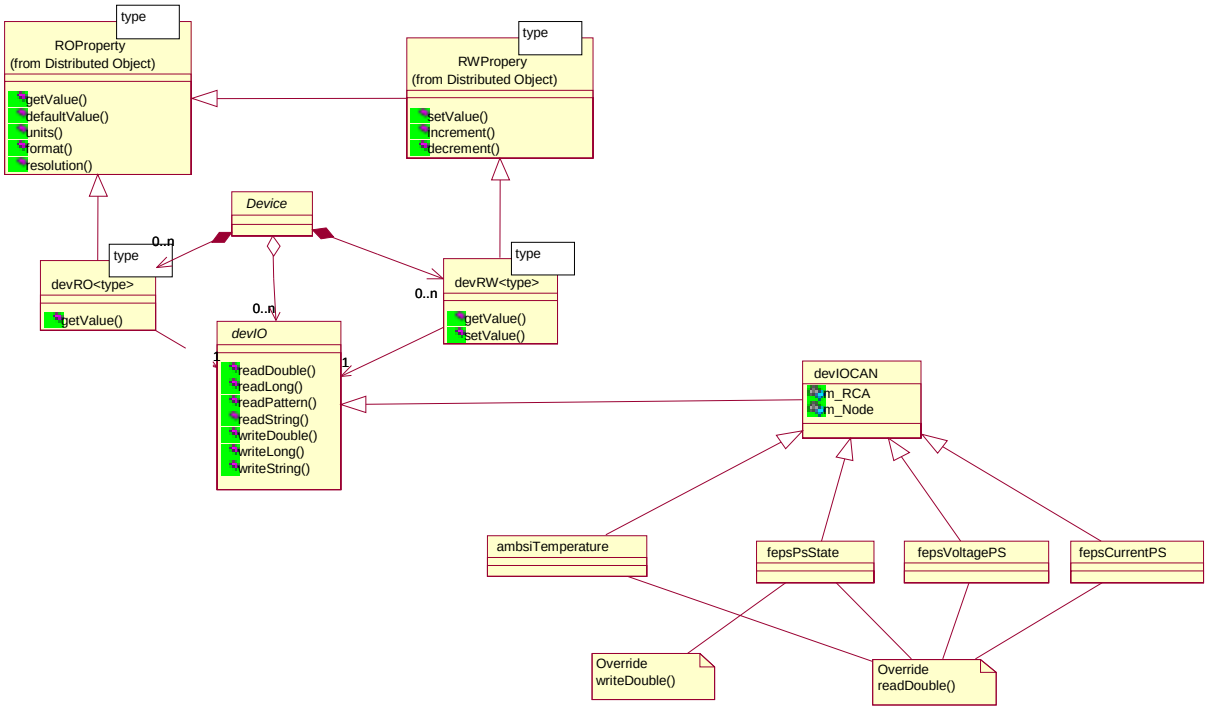


Figure 10 - CAN Property Inheritance

This section describes properties that access hardware monitor and control points. Read or write operations on such properties generate I/O activity on an external bus, such as the AMB CAN bus, or an RS232 serial port. The design is described for all external I/O methods, with additional detail for the case of AMB CAN properties.

Device properties are subclasses of the standard ACS property classes. Read-only properties correspond to hardware monitor points, and read-write properties correspond to hardware control points. Values read or written via the properties are typed values, and the underlying implementations provide, transparently to clients, all necessary data conversions and I/O access methods.

The implementation of device properties is separated from the interface. Device property implementation is provided by the abstract "devIO" class. A reference to an implementation object (from the device property object), allows the property to access the external bus using typed values whenever a monitor or control point is used by a client. This allows device properties to have a single interface to the implementation, regardless of I/O method or property type. In general, the implementation classes fulfill two requirements: data conversion (between data format on the bus, and typed values in the LCU), and I/O access. In many cases, the implementation class may be entirely an interface adapter, placed between the LCU and the external bus access classes.

Henceforth, this section specifically describes CAN properties, as a particular type of device property. CAN properties are those that access monitor and control points on the AMB CAN bus. CAN properties are configured by a device (of which they are a part) using the configuration database. In addition to the characteristics provided by the ACS properties, CAN properties have a "relative CAN address" characteristic. Note that the CAN node address (or, equivalently, the serial number) is not a characteristic of the property; rather, it appears as a characteristic of the CAN node that is a part of the device. Devices "owning" CAN properties must instantiate the CAN node object that provides the node address, and, in turn, the full CAN address of the property. Thus, devices of this type have a "node serial number" characteristic for each CAN node required by the device. The node serial numbers are converted to node addresses when the node objects are instantiated by the device (c.f. CAN access package).

The devIOCAN class provides default data conversion routines for all property types. Objects of this class access the AMB CAN bus using the ambCANNode or ambSimpleCANNode classes provided by the CAN access package. The devIOCAN class may thus be viewed as an interface adapter class, as it simply uses an ambCANNode object to do the I/O, and converts the data format between the CAN bus and the LCU.

When the devIOCAN class does not provide the correct data conversion, a subclass of devIOCAN may be used. Typically, such a subclass need only override a single method from devIOCAN since the subclass is normally highly specialized to a specific property. The canTemperature class (which allows access to the DS1820 temperature sensor on the AMBSI1 board) is an example of such a specialized subclass of devIOCAN.

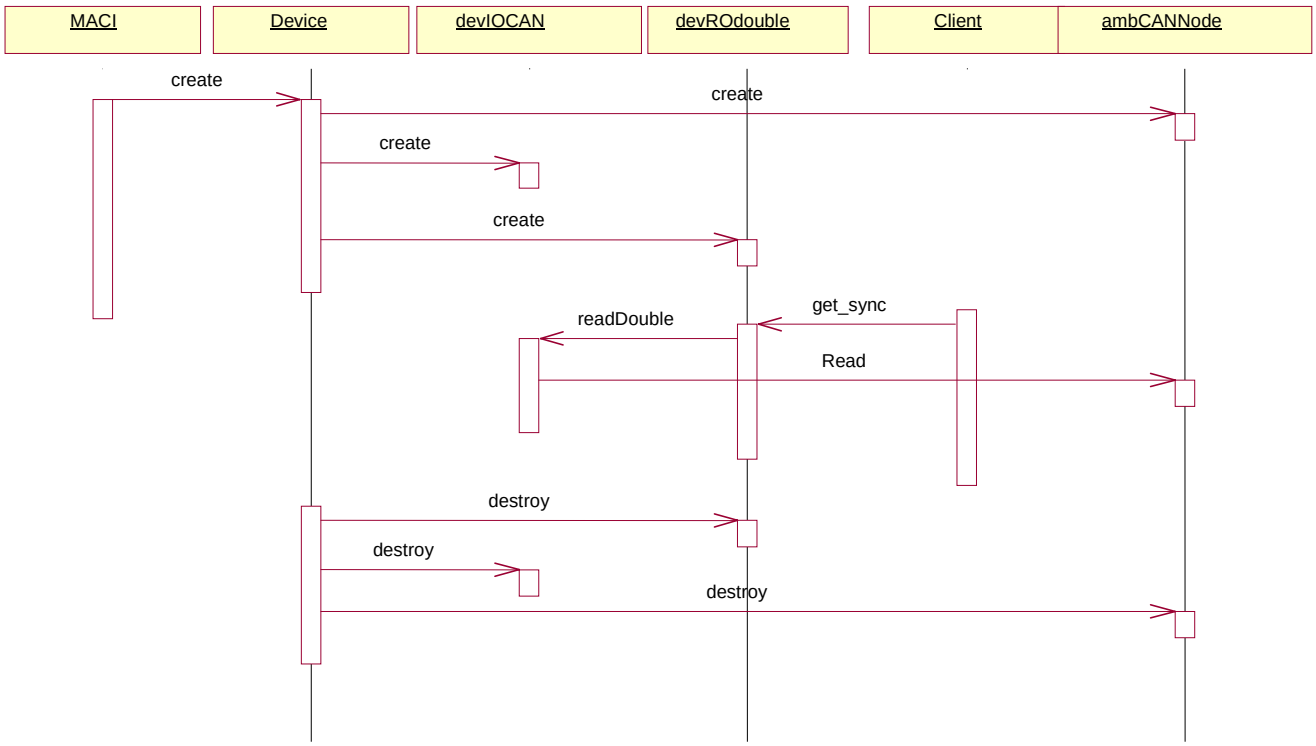


Figure 11 - CAN Property interaction

11.4 Other Device Interfaces

M. Brooks
Last Changed: 2001-05-05

In addition to the AMB it is expected that there will be a small set of unique interfaces to be supported. This may occur due to the use of commercial off-the-shelf components or designs re-used from other astronomical instrumentation.

The following interfaces are envisaged:

- GPIB: Where a need for GPIB interfaced devices exists, a GPIB bus would be run to a COTS GPIB card in the nearest real-time computer (ABM or ARTM). A layer of software is required, similar to the CAN access package, in order to connect device objects to GPIB devices. If Properties are needed for these interfaces it is TBD whether a new Property will be derived (as for CAN) or whether Logical (memory-based) Properties will be used (with the memory location being written by an application).
- Ethernet: Several devices, such as the ACUs, will have Ethernet interfaces for auxiliary reasons such as debugging access, or static parameter configuration. Where a need exists for access to a device for monitor and control via Ethernet, standard ACS services should be used. Since no ACU monitor and control will occur via Ethernet, the ACUs will not be required to use ACS services.

11.5 Monitor and Control Points

B. Glendenning
Last Changed: 2000-10-23

In software, a Monitor or Control point is represented by an object. They have the following features:

- Each one has a unique name. The name is human readable, and follows the actual hierarchy that gives the "location" of the point. For example: *antenna:.compressor.ambient_temperature*.
- Some details about a monitor point, e.g. its precision, are loaded from the configuration database described previously.
- Monitor points are read-only and Control points are read-write. The "read" of a control point will retrieve its last commanded value.
- While they typically correspond to an actual hardware value, some will be purely synthesized in software.
- In type they are either a scalar value (Double, Integer, String, Enum), or a Sequence (1-D array) of scalar values. The type of a Monitor or Control point is fixed at compile time.
- They are always embedded in one and only one software Device (see next section).
- They are based upon ACS Property classes (possibly without change).

11.6 Software Device Conventions

F. Stauffer
Last Changed: 2001-10-31

11.6.1 Description

A *Device* is an idealized description of a logical or hardware device in the system. If the hardware device is an 'ideal' device, then the *device* is just a container for the monitor and control points. Otherwise, the *Device* creates the 'ideal' device on top of the hardware. *Devices* are either low level, such as a software representation for a hardware module, or they are high level and composed of other devices such as the electronics comprising the signal chain - front end, downconverter, LOs, and digitizers. The low level *devices*, referred to as Device Controllers, will run on the real-time machines and possibly are synchronized to timing events. The higher level *Devices*, referred to as Composite Devices, may run on either the ACC or the real-time machines.

Sometimes devices do not map exactly onto the hardware. For example they may have an interface that accepts a time polynomial rather than requiring more rapid commanding. Similarly, they may canonicalize the data types they present to the rest of the system. Direct control of the hardware device through CAN is of course possible for debugging or other purposes.

It is important to note that we intend that devices to not normally require real-time commanding. This has these consequences:

- Device controllers perform time-synchronized actions.
- Device controllers are on the real-time machines.

- Commands must be sent in advance to the *devices* along with the time at which they should be activated.
- Device controllers and not the calling program handle fast control. This can be accomplished by sending the device controller high-level commands (*e.g.*, polynomials or tables) that last for some time and letting the device controllers control the hardware at faster rates. This is described further in the Device Controller section.

Device is an abstract base class from which particular device controllers and composite devices will be derived. The *device* class is implemented on top of the ACS Distributed Object. In brief, it contains read-only ("monitor") and read-write ("control") Properties, and possibly logical Properties. Each Property in turn will be associated with a number of Characteristics that describe it (for example, the units). A *Device* also has Characteristics such as name and description, and possibly other commands (in particular, multi-argument methods) that do not naturally fit into Properties.

11.6.2 Commands

The only standard commands shared by all devices are those related to changes in the standard states (see next section).

11.6.3 States

Device behavior is modeled with a statechart, Figure 13. When a composite device performs a state transition, it will pass the state transition to the devices they contain. Devices inside of a composite device can transition to other states without necessarily causing the composite state to make a state transition. For example, this means a composite device can be ENABLED and have a DISABLED device.

It will be useful for these states to be available as Properties for monitoring and other purposes. Methods of accomplishing this are under discussion.

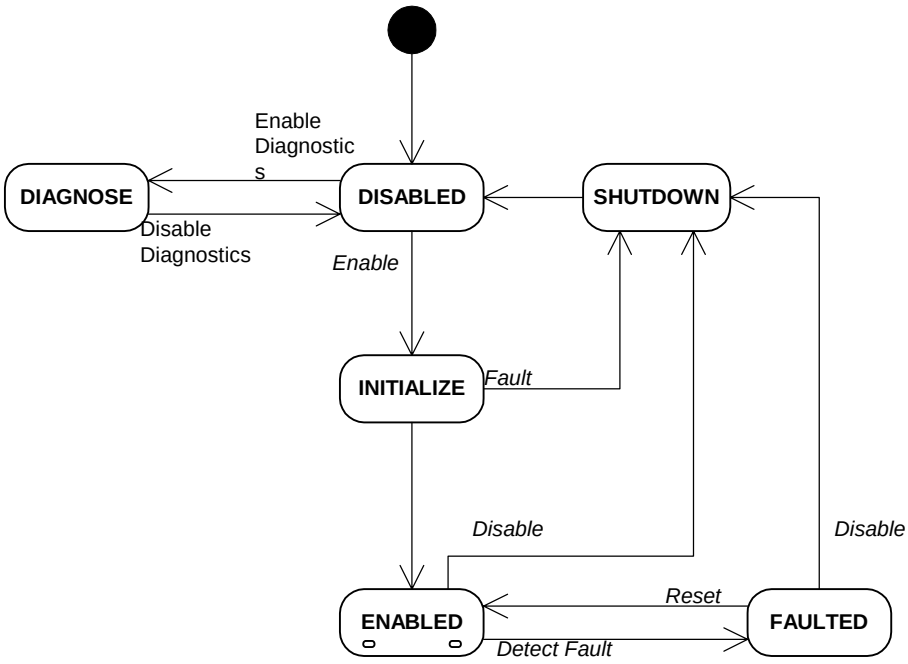


Figure 13 – Standard states shared by all devices.

As shown in the figure all devices must be in one of the top-level states:

DISABLED The equipment is in power-off, low power, or some generally inaccessible state. DISABLED only accepts commands to transition to other states and monitor commands.

DIAGNOSE State used to debug the hardware.

ENABLED In this state the device accepts M/C or other commands. Commands that are long running (for example, a nutator program) are executed as parallel running state machines inside the ENABLED state.

INITIALIZE Perform actions to ready the hardware, such as enable power, calibration, etc.

SHUTDOWN Perform actions to disable the hardware.

FAULTED A device usually enters this state "spontaneously" from ENABLED when a fault condition is detected, either by the device hardware or by software inside the Device. A fault condition is faulty hardware behavior that needs intervention to correct or a time tagged command that is late. This state accepts M/C or transition commands. Some conditions may exist that cause alarms, but they are not severe enough to cause a fault. FAULTED can only be exited by human intervention.

11.6.4 Software Interface

11.6.4.1 Class Diagram

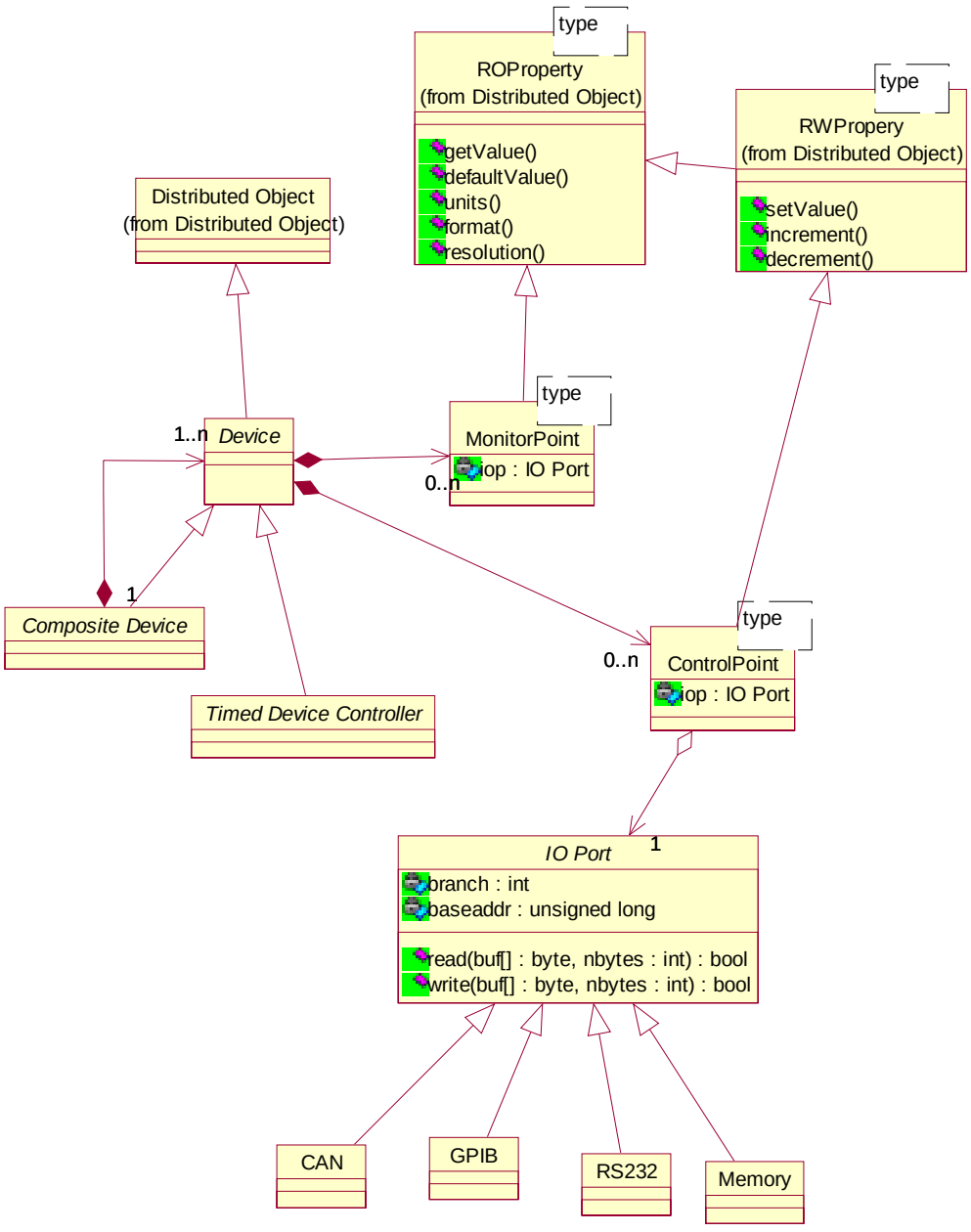


Figure 14 - Fundamental Device and Property Classes

11.7 Device Controller

F. Stauffer
Last Changed: 2001-10-31

11.7.1 Description

The device controller is the lowest level software device, just above the hardware, and runs on the real-time machines. Optionally, device controllers can have timing-event synchronized actions, long running actions, or control loops to accurately set and track device properties. These device controllers have the control software implemented as state machines inside the ENABLED state.

11.7.2 ENABLED Sub-States

11.7.2.1 Control Loops and Long Running Commands

An example is a device that tracks a settable value, for example the sub-reflector focus.

11.7.2.1.1 States

The device specific sub-states are usually IDLE and EXECUTING. Control parameters can be updated while in the EXECUTING state.

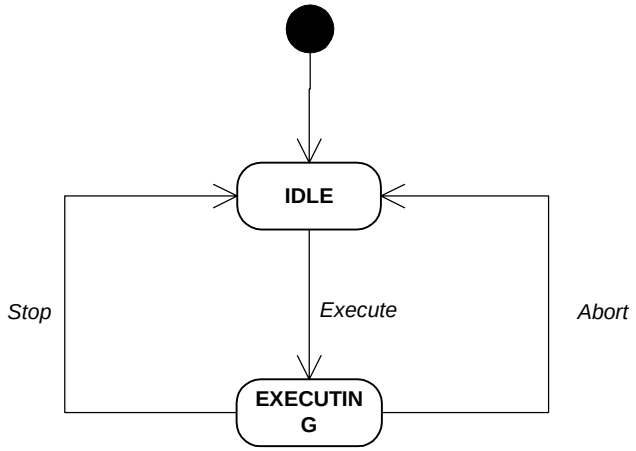


Figure 15 - Standard states shared by control loops and long running sequences.

IDLE In this state the device accepts M/C or other commands. Commands that are in the IDLE state should not be long running commands.

EXECUTING The device controller is executing some (potentially) lengthy command or tracking a set-point. The Stop command ends the state gracefully (e.g., at the end of the next controlled update). The Abort command is intended to stop it immediately.

11.7.2.2 Precise Timing Control

ALMA has some devices that require time-synchronized monitor and/or control. These devices must meet the ALMA hardware timing specification.

For monitoring purposes, we assume that the monitor point get method will supply a time-tag. The time-tags will correspond to the time the monitor point was sampled in the hardware. That is, it will be corrected back to the timing event that generated it, including any device-dependent offset from the timing event.

11.7.2.2.1 States

For commanding purposes, the model is that when in an IDLE state the long-running command is set up (it might involve interaction with many Properties). The time at which the command should start is set (through another property) at which time the device controller transitions to the ARMED state. The device transitions to the EXECUTING state at the time requested. The EXECUTING state indicates that a long-running command is executing. It will often have device-specific sub-states. To change the long-running command, the device controller is first returned to the IDLE state. An example of a time-synchronized program is nutator positioning.

Time-synchronized action sequences usually are modeled as a device-specific list of control point value and dwell time entries to be executed in some device, such as a nutator sequence. Time-synchronized hardware programs run in hardware if supported, or in the device controller itself if not.

The list is started with respect to the timing events, and the dwell times are the time each entry is active. The time each entry starts is the start time plus the sum of dwell times since the start. Dwell times are expressed in milliseconds. This is based on the need for flagging total power data at 2 ms. Device programs will generally be chosen to fit evenly into some number of timing event periods.

The list is executed in sequential order, and when the last entry is completed, the list wraps around repeating the program until stopped. A list entry is executed by setting the control points and waiting for the dwell time to expire.

To ensure the EXECUTING state executes correctly, control points will have an extra characteristic that indicates if the control point may be set while EXECUTING. This allows for feedback control, but prevents control that will break EXECUTING. For example, setting gain values will not break the state, but setting the controlled position asynchronously is not allowed.

When EXECUTING, and an error occurs or the user wishes to disable the device, a transition from EXECUTING occurs. The UML statechart model specifies that transitions on the enclosing state implies transitions from the lowest level nested sub-states first. The current sub-states are not remembered and not re-entered later because timing synchronization could be violated. When the EXECUTING state is re-entered, sub-states need to start from IDLE**.**

Time-tagged commands that have not been sent sufficiently in advance to meet the underlying timing specification will result in a transition to the FAULT state.

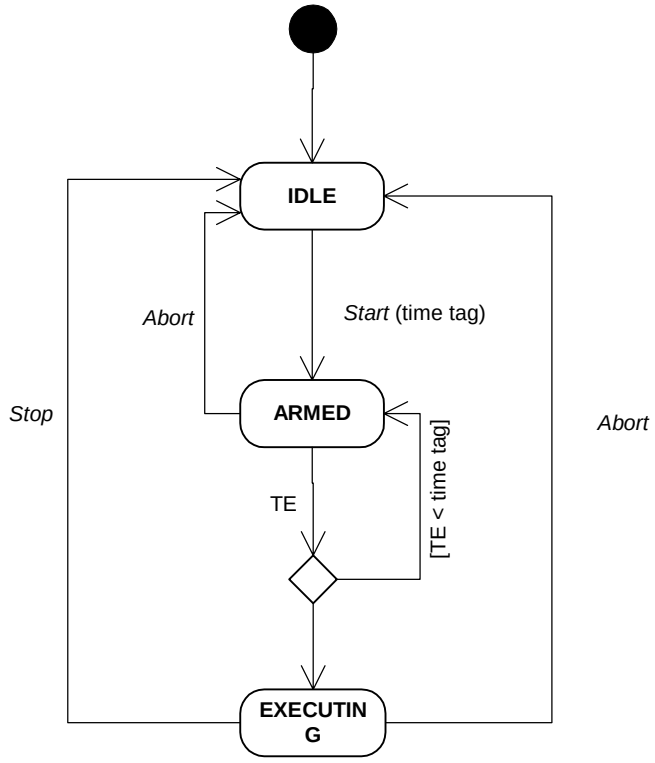


Figure 16 - Standard states shared by all timing event synchronized control sequences.

IDLE In this state the device accepts M/C or other commands. Commands that are acted upon immediately result in the device controller remaining in the IDLE state. Commands that are long running (for example, a nutator program) are executed in the EXECUTING state.

ARMED The caller has indicated that the device should start EXECUTING. This state waits until the time, at which the device should start EXECUTING. ARMED is used only when time synchronization is needed. If precise timing is not needed, it is TBD whether the ARMED state will be removed or transitioned through "instantly."

EXECUTING The device controller is executing some (potentially) lengthy command. This state will often have additional device-specific sub-states. For example, a nutator while running a nutator sequence might be in an OFF, ON, or MOVING sub-state. The Stop command ends the state gracefully (e.g., at the end of the next nutator cycle). The Abort command is intended to stop it immediately.

11.7.2.2.2 Timing

The device controller is responsible for providing a time tagged interface such that the user and the communication link do not have to be real-time. The commands are sent with a time tag to specify the start timing event far enough in the future that the worst-case latency is allowed for. The ENABLED sub-states with timing synchronization are ARMED, wait for the start timing event, and transition to EXECUTING. The EXECUTING state executes the time-synchronized control.

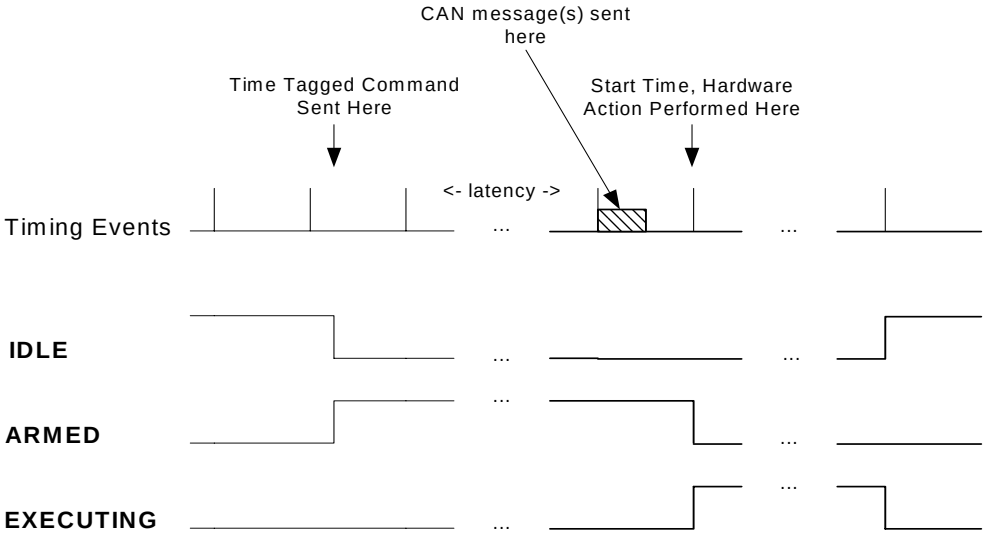


Figure 17 - Timing diagram example for a time synchronized hardware control.

11.7.3 Software Interface

11.7.3.1 Class Diagram

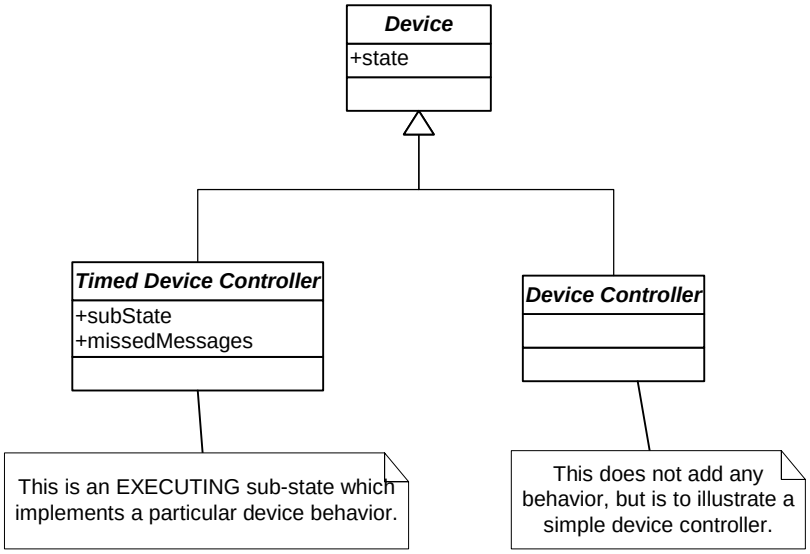


Figure 18 - Device controller class diagram.

11.7.4 Implementation

The described behavior presented in the previous discussion is from the user's point of view, and it represents the hardware. Software control is modeled with the same state diagrams. The major difference is that the time synchronized EXECUTING sub-states are phase shifted some number of timing events ahead of the hardware. For CAN devices, the phase shift is one timing event. Software control is synchronized by timing events, and the control messages reach the hardware in the timing window before the hardware acts on it. The timing diagram in the next figure shows the relationship between the software and hardware states for CAN devices.

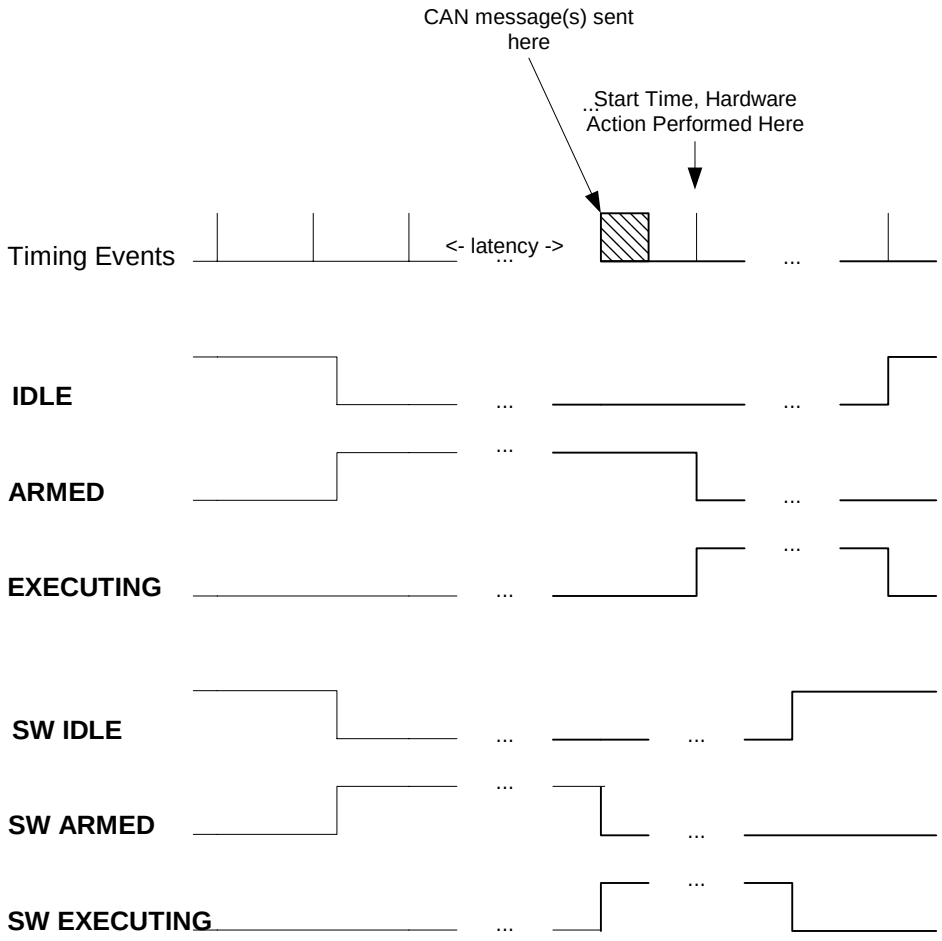


Figure 19 - Timing diagram example for time synchronized hardware compared to the software control. For devices that are not time synchronized, the ARMED state is not used and the transition is from IDLE to EXECUTING.

11.8 Device Creation and Management

B.E. Glendenning

Last Changed: 2001-05-01

TICS follows the ACS philosophy for device creation. In particular, the *Manager* is used to obtain references to distributed objects, and the *Activator* is used to create them. Subordinate parts (M/C points, sub-devices) of a particular *Device* are constructed indirectly. In particular, their construction is triggered by asking the *Manager* for an object reference to the subordinate part, which ultimately causes the constructor for that subordinate part to be called. This indirection allows for the substitution of a subclass that conforms to the required interface (for example, for simulation), or to transparently change the computational host that runs the subordinate object without having to make any change to the "outer" *Device*.

Part of the activation protocol will include the framework making available to the *Device* under construction an IDL interface to the ACS configuration database. The *Device* will then bootstrap itself by reading its required configuration parameters through the interface.

11.8.1 Python Issues

An ACS *Activator* will be provided to allow Python *Devices* to be implemented within the ACS distributed object framework.

It is clearly desirable for these Python *Devices* to have Properties directly, however it may not be feasible to have a direct Python *Property* implementation available on the desired timescale. As an expedient, a factory will be provided that can create *Logical Properties* (which are implemented in C++) on behalf of the Python *Device*. A *Logical Property* merely reflects the value of a memory location. It may be set via the *Property.set()* method if the property is read/write, otherwise it may be set through the ACS database interface, which will have an IDL binding and is hence accessible from binding. In this fashion a Python *Device* may have *Properties* whose values are defined in Python. While this scheme requires an extra inter-procedure call, the overhead should be acceptable from the scripting language until a native Python implementation is available.

An alternative implementation would be to introduce a *Callback Property* class that merely forwards its I/O request to a callback object. This callback would be defined in IDL and hence could be implemented in Python. The Python *Device* would merely insert its own callback into the *Callback Property*. These two possibilities require further investigation.

In addition to the database interface, the ACS *Logging* interface and (at lower priority) an interface for the ACS Error generation will be provided.

11.9 Mount

R.W. Heald

Last Changed: 2001-02-07

11.9.1 Description

The Antenna Mount System (AMS) controls the movement of the ALMA antenna for astronomical observations and testing. All ALMA antennas use identical hardware and software.

The AMS logically sits between the Array Control Computer (ACC) and the Antenna Control Unit (ACU). The ACC commands the antenna position by sending tracking commands

consisting of a *target* and optional *pattern* to the AMS. The AMS combines the *target* and *pattern*, converts them to horizon coordinates useable by the antenna, applies the appropriate corrections, and sends the results to the ACU. This causes the antenna to move in the desired manner.

Trajectory commands to the ACU have 4 parameters consisting of the elevation and azimuth positions, and the elevation and azimuth velocities. A zero elevation points the antenna at the horizon, and a zero azimuth points it due north.

For each antenna axis, the AMS has two positions (called "limits") that define the extremes of antenna movement for that axis. An attempt to exceed a limit causes antenna axis movement to stop at the limit and a software alarm to be produced.

The antenna normal range of azimuth motion is 270 degrees on either side of north. This implies there are two possible azimuths to reach all southern sky regions. Tracking commands to the AMS specify the region to be used, either the "+" or "–" azimuth region, or the closest (the default).

The antenna normal range of elevation motion is 2 to 125 degrees. This means there are two possible elevations to reach the sky region above 55 degrees. Tracking commands to the AMS specify the region to be used, either 2 to 90 degrees (the default), or 90 to 125 degrees (called "over-the-top").

When the AMS is given a *target* that is below the elevation limit the AMS moves the antenna to at the correct azimuth at the elevation limit. There it tracks the target in azimuth only until the *target* rises above the elevation limit. The AMS works in this manner for both normal "front side" elevations, and "over-the-top" elevations where the lowest elevation limit is 55 degrees above the horizon. Notice some (especially those with a optical astronomy background) consider this behavior to be strange, and it will be reviewed for the ALMA.

Tracking at the sidereal rate is not possible within a 0.2 degrees radius of the zenith. When tracking within this region the AMS moves the azimuth as fast as possible to resume tracking the source when it leaves this region. There is no guarantee of good tracking while the antenna is within this region.

The AMS can cause the antenna to follow a *pattern* superimposed on the *target* being tracked. The speed along the *pattern* is constant and settable, and the movement is synchronized between antennas. A pattern can be described in equatorial or horizon coordinate systems. Available *patterns* include a simple offset and raster scan. The lines of the raster scan can follow the right ascension, declination, azimuth, or elevation coordinate.

Patterns are composed of a series of separate, short *strokes*. For the raster scan pattern a *stroke* is a single line of the pattern. Calibration procedures can be performed between the strokes. Antenna motion can be specified to begin and end with any given *stroke(s)* within the *pattern*. Movement along a *stroke* is continuous and atomic.

The AMS can perform a drift scan. In a drift scan the antenna is moved to a position preceding the track of the desired object and is then held stationary to allow the object to track through the antenna's boresight.

The AMS constantly reads the actual positions captured by the ACU and compares this with the commanded position to determine the error on the sky. Comparing this error with a tolerance determines whether the antenna is “on source” or “off source”. The AMS takes the most pessimistic view in this determination. That is, the AMS must find two consecutive positions within tolerance before “on source” is declared, and when a position is found outside the tolerance both that position and the previous position are declared “off source”. The AMS reports the results to the ACC whenever there is a change. The tolerance value is settable.

The AMS has an On-The-Fly (OTF) mode. When in OTF mode, the AMS appends a time stamp to the positions obtained from the ACU and sends them to the ACC.

The OTF observing mode will support the relevant antenna coordinate system. The approach is to put a rotation into the horizon coordinate frame, such that the equator of the system goes through the source (the transmitter on its tower). This is still a conventional spherical system, but with a single rotation, the same as, but simpler than, the rotation from equatorial to Galactic coordinates, for example. A simple matrix will take care of this rotation.

The modified OTF raster scanning is a special case of more general non-rectangular scanning schemes. For example, making the scanning length to be +/- $\sqrt{R^2 - Y^2}$ where R is the defined radius of the parent "square map", i.e. R=X/2, X is the horizontal extent of the map, and Y is the distance of the current scan from the center of the map. In this definition X is measured horizontally and Y is measured vertically, however, it could apply to any rotation of the scanning direction.

11.9.2 Properties

- Status (Off, Stopped, Moving, Error)
- On/Off source position
- On the sky error tolerance for On/Off source position
- Azimuth and elevation wrap
- OTF mode
- Commanded equatorial position
- Commanded horizon position
- Actual horizon position
- Remaining track time for source
- Current UTC time
- UT1-UTC
- Sidereal time

- Polar motion coordinates
- Antenna motion limits
- Antenna site location
- Observing frequency
- Ambient temperature
- Atmospheric pressure
- Relative humidity
- Temperature troposphere lapse rate
- Pointing model coefficients

11.9.3 Commands

- Track a celestial object or move to a stationary position

  `track(system,coord1,coord2,pm1,pm2,velocity,parallax)`

  where `system` is either `equatorial, ecliptic, galactic,` or `horizon`

  `track(name)`

  where `name` is a source from a catalog or a special position such as `stow`, etc,

- Perform a pattern stroke

  `stroke(type,system,arguments)`

- Add a horizon coordinate offset

  `offSaa(az,el)`

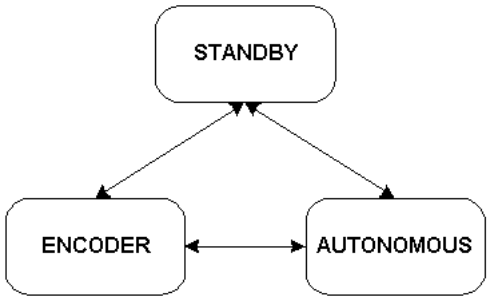
- Add a equatorial coordinate offset

  `offSad(ra,dec)`

- Stop all antenna movement

  `stopMotion()`

11.9.4 States



Sub-States of the Enabled State

11.9.5 Timing

Trajectory commands and position monitor requests are sent over the CAN bus to the ACU after each *timing event*. Their delivery is time critical and must be in accordance with the timing specifications given by Brooks and D'Addario (2001).

11.10 Test Correlator

J. Pisano
Last Changed: 2001-09-17

11.10.1 Description

The ALMA test correlator device is a "configure and run" time-synchronized device, that is, it is configured for an observation and then commanded to start correlating. Raw lag results flow from the test correlator hardware to the correlator control computer (CCC) and then processed to spectral data sets and ultimately transmitted via the science data pipe to the Data Collector. For the TI, the Data Collector will be the ACC. Unlike other devices in the TI, the test correlator device communicates to the ACC via a switched Ethernet connection.

The test correlator device is configured for observation by setting the following items:

- The correlator dump time in units of 1.31072 ms correlator ticks.
- An integration time which defines the number of correlator dumps to accumulate internally before transmitting the results to the Data Collector.
- Bin switching parameters – number of bins and dwell time.
- Data processing options defining how the lags are converted to spectral channels including geometric delay parameters for interferometric observations.
- The correlator system mode which sets a group of properties which are outlined in the following table.

CORRELATOR MODE ID	BANDWIDTH	POLARIZATION PRODUCTS[5]	LAGS	DELAY RESOLUTION	DELAY RANGE
1(cross-products)	800 MHz	0R X 1R 0L X 1L 0R X 1L 0L X 1R	512 Leads & 512 Lags	5 ns	10 μs
2(cross-products)	800 MHz	0R X 1R 0L X 1L	1024 Leads & 1024 Lags	5 ns	10 μs
3 (self-products)	800 MHz	0R X 0R 0L X 0L 1R X 1R 1L X 1L	1024 Lags	N/A	N/A
4(cross-products)	100 MHz	0R X 1R 0L X 1L 0R X 1L 0L X 1R	4096 Leads & 4096 Lags	20 ns	80 μs
5(cross-products)	100 MHz	0R X 1R 0L X 1L	8192 Leads & 8192 Lags	20 ns	80 μs
6 (self-products)	100 MHz	0R X 0R 0L X 0L	8192 Lags	N/A	N/A
7 (self-products)	100 MHz	1R X 1R 1L X 1L	8192 Lags	N/A	N/A

[5] This defines which antenna – polarization product is supported for a given mode where antennas are defined by **0** or **1** and polarizations are either **R** or **L**.

Once configured, the test correlator hardware is commanded to start an observation at a predetermined timing event. Correlator chip sub-integrations are set to be either synchronized or not to the array-wide 48ms timing events. At each dump, raw lags are extracted from the test correlator hardware, any data processing options set in the configuration stage are applied, fine delay corrections are made, and the data are added together in an integration accumulator. Once an integration has completed, the spectral results are tagged with a header identifying each polarization product and sent to the Data Collector.

11.10.2 Properties

Although each property read-only (RO), they are set via commands as function parameters:

- Observation control properties
  - Correlator dump time – an integral number of 1.31072 correlator ticks.
  - Integration time – an integral number of correlator dumps.
  - Integration duration – an integral number of integrations.
  - Number of correlator bins – a integer of value 1, 2 or 4 which allows integrations at the correlator chip level to switch among the specified bins at specific 48ms timing intervals [6].
  - Correlator bin switch time – an integer value which specifies the number of 48ms timing intervals devoted equally to each bin. This value is ignored if the number of correlator bins is 1.
  - Correlator mode – an integer 1 – 7.
  - Delay model coefficients for interferometry modes. The form of the delay model is: $\mathbf{d\tau(t) = D_0 + D_1 t + D_2 t^2 + D_3 t^3 + D_4 t^4}$
- Data processing properties (which are performed in the following order)
  - Boolean flags specifying that the following should be done:
    - Van Vleck Correction
    - Hanning Windowing
    - FFT
  - Spectral Averaging specifying how many adjacent channels to average together
  - Select a subset of spectral channels
  - Decimation of spectral channels by specifying every N-th channel to keep

11.10.3 Commands

Controls:

- Set observing properties via a single function.
- Set data processing properties via a single function.
- Set the geometric delay parameters via a single function.
- Start observing on a specific timing event for a specified number of integrations having the correlator chip sub-integrations either synchronized or not to the timing events.
- Stop observing at end of current integration.

[6] This feature allows switching (frequency or beam) on 48ms timing events.

- Abort observing – note that this allows for a partial integration to be sent.
- Reset – perform a cold or warm reset on either the test correlator hardware and/or the CCC.
- Run a diagnostic test on the test correlator hardware. This must be performed when the test correlator is not running (correlating)

Monitors:

- Current test correlator device state
- All of the following monitor points are read-only:

Quantity	**Units**	**Resolution**
+ 5 VDC	Volts	5 mV
- 5 VDC	Volts	5 mV
-2 VDC	Volts	5 mV
+24 VDC	Volts	15 mV
+15 VDC	Volts	10 mV
-15 VDC	Volts	10 mV
Corr. Rack Temperature	Degrees Celsius	0.1 degree

- Any of the following test correlator characteristics all of which are read-only:

Index	*Item*	*Description*
1	***Cross-correlation tick time***	***Number of milliseconds of one correlator tick in cross-correlation mode. This value is 1.30172.***
2	***Auto-correlation tick time***	***Number of milliseconds of one correlator tick in auto-correlation mode. This value is 1.30172***
3	***Maximum data rate***	***The fastest rate at which data can spew out of the correlator in units of results/second. Raw lag results will be 32-bit integers while spectral points will be 32-bit floating-point numbers***
4	***Bandwidth modes***	***A list of the available bandwidth modes. These are 800 MHz and 100 MHz. An asterisk next to the mode indicates that a Xilinx FPGA personality image needs to be downloaded for this mode, e.g., "100MHz*".***
5	***Mode Change Time***	***This describes the time (in seconds) it takes to perform a correlator mode change. For 800MHz mode, the value is 100 ms and for narrow band mode (100MHz) the value is 40 seconds.***
6	***Minimum Dump Time***	***This is the shortest dump time (in milliseconds) for the correlator hardware. This value is 48 milliseconds for***

		synchronized dumps and ~70 milliseconds for non-synchronized dumps
7	***Computer Type***	***This is a simple string identifying the correlator computer type. It is "Motorola MV2700 MPC750".***
8	***Monitor point list***	***A list of monitor points names.***

11.10.4 States

The test correlator has no sub-states beyond the standard top-level states used for devices with precise timing requirements as described previously.

11.10.5 Processing flow

The following sequence diagram shows the processing flow of correlator configuration and data processing.

The primary actors are the ACC and Data Collector while the test correlator is a secondary actor. The main internal objects are represented by the classes:

- CC_Manager which is responsible for coordination of internal execution and interfacing to the ACC.
- IntegrationManager which is responsible for configuration and control of the test correlator hardware.
- CorrelatorManager which provides a hardware abstraction layer to the test correlator hardware.
- DataProcessor which is responsible for extracting raw lags from the test correlator and processing them to spectral results by applying the selected data processing items, e.g., Hanning windowing, VanVleck correction, FFT, etc. For interferometry modes, fine delays are applied to the spectral results after the FFT is applied to a correlator dump. Finally, DataProcessor is responsible for assembling the "bricks" of spectral results and delivering them to the Data Collector.

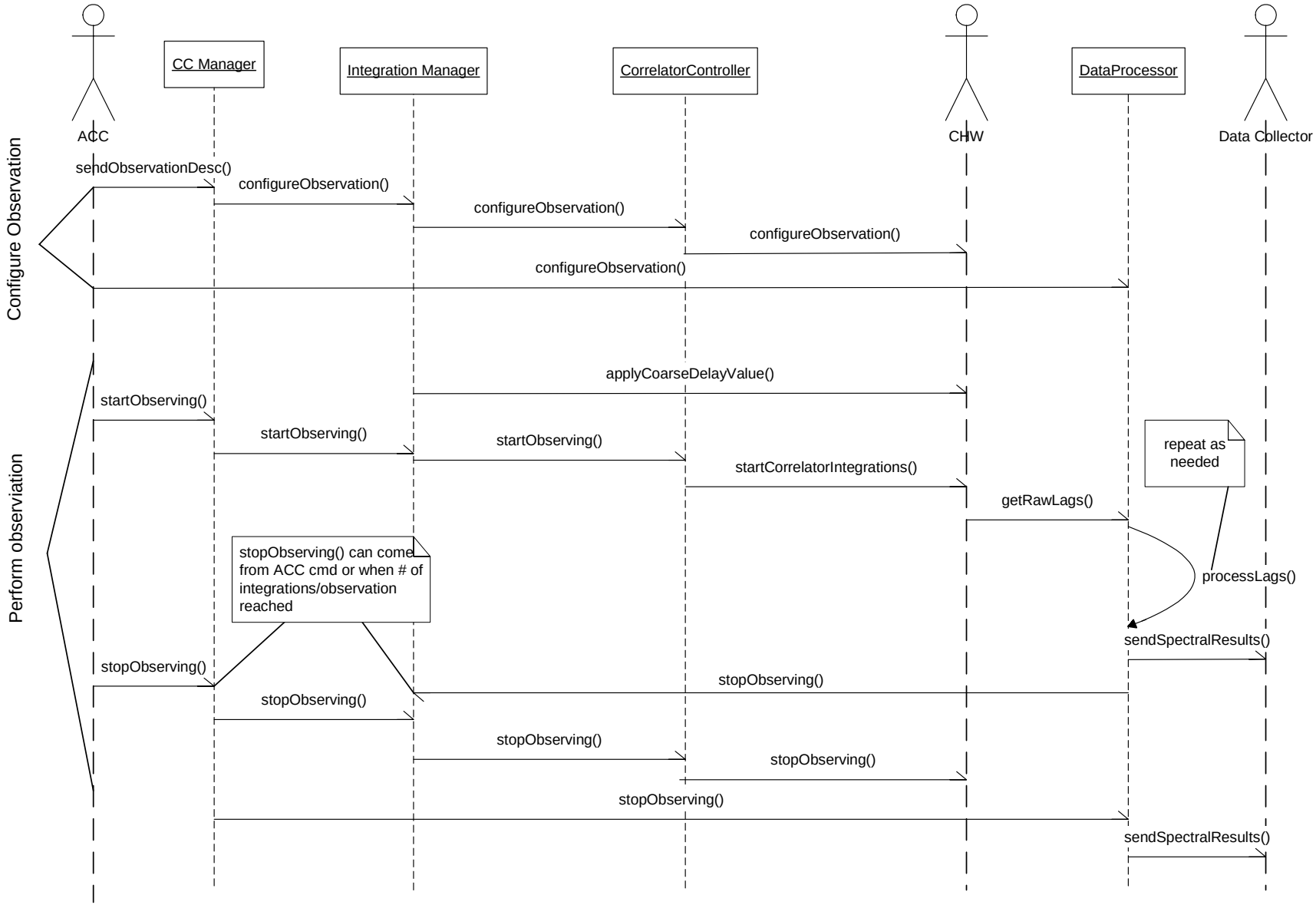


Figure 20 – Correlator Control Sequence Diagram

11.10.6 Timing

Starting and stopping correlator dumps must occur on a timing event. The specific timing event is specified in the start (or stop) observing command. In order for this to occur, the commands must be sent from the ACC far enough in advance to deal with network latency and the latency of translating the start (or stop) observing command to a command understood by the test correlator hardware so that it begins dumps on the specified timing event. This latter latency value must is at least 72ms.

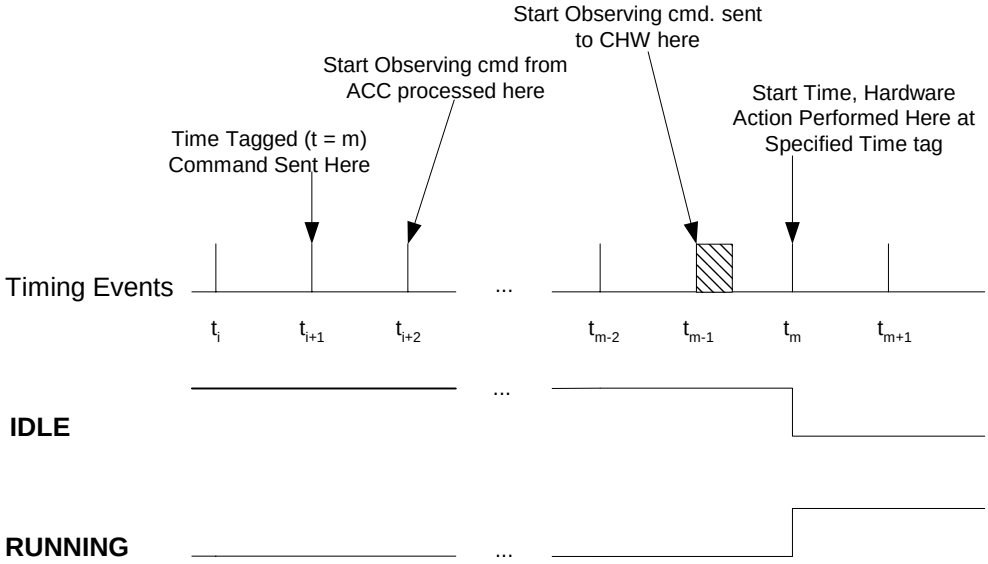


Figure 21 – Starting and stopping correlator dumps on a specific TE

For interferometry mode, the coarse delay which is derived from the delay model evaluation, is applied before the test correlator is commanded to start observing.

11.10.7 Class Diagram

The following figure provides a class diagram overview of the TestCorrelator device. It contains correlator configuration and data processing information as previously described. Also included are the various monitor points.

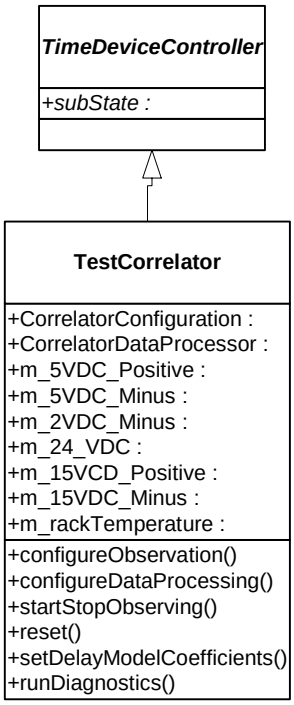


Figure 22 - Class diagram for the test correlator device

Figure 23 shows the CorrelatorDataProcessor class which holds the specific data processing configuration items and populates a CorrelatorScienceDataResults object which encapsulates the header information plus spectral data for each correlator integration.

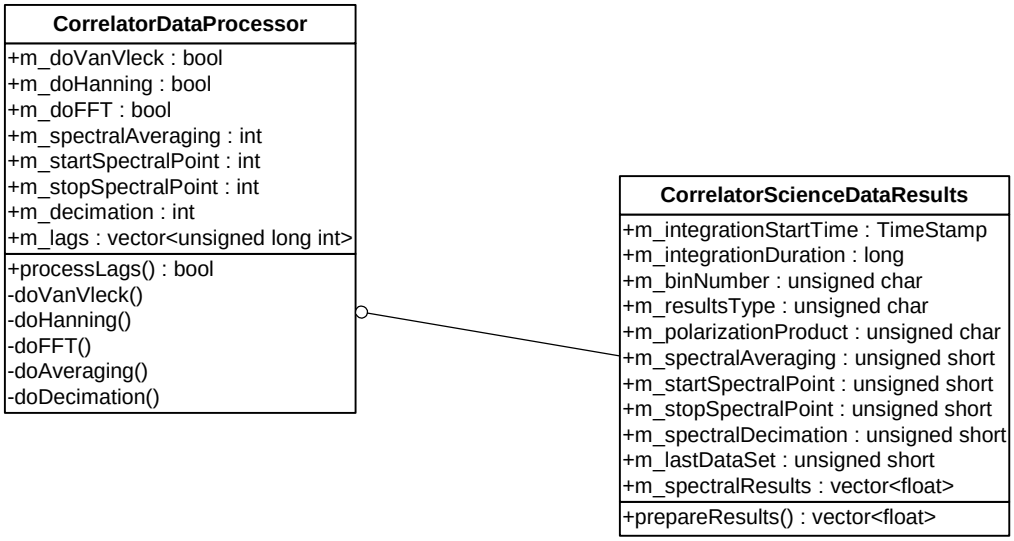


Figure 23 Correlator Data Processor

11.11 Nutator

F. Stauffer
Last Changed: 2001-10-31

11.11.1 Description

Details related to the internal (embedded) software view of this device are not covered. *This device is currently under design so many details are still TBD.* The fundamental specification of the nutator is that it can carry out an ON/OFF cycle at 10Hz with a 10ms switch time (i.e., 80% duty cycle).

The nutator hardware is assumed to have a programmable list of target position, move time, and dwell time triples – typically there will be one ON position and one OFF position, however there is no practical limit to the number of target positions that can be defined. The target positions can be arranged in a program sequence (e.g., ON→OFF) that can then repeated by the nutator until stopped or aborted. The move time and the dwell time are used to flag data bad or good, and the nutator hardware verifies if the times are achievable.

The nutator follows the general ALMA synchronization philosophy outlined in ALMA Memo #298. In brief, that time-critical control points are set in advance of a timing event (48ms period), which strobes the set value active.

The nutator sequence is programmed in 1ms increments that are aligned with the system-wide timing events to some TBD accuracy. The nutator has an active program and a standby program. The standby program can be loaded while the active program is running. The standby program is loaded as the active program when ready to run. In normal operation we assume that the programmed sequence is achievable and monitor the nutator state relatively infrequently – probably once per timing event. However a nutator position monitor will be available which can be called at up to the CAN bus bandwidth (~2000 times per second) for debugging or other special purposes.

This description assumes a one-axis nutator programmed with a fixed series of positions. For the ALMA array a more elaborate nutator model may be required (for example, the hardware will be capable of spiral patterns).

11.11.2 Properties

Parameters are assumed to be set infrequently - generally just before the nutator program is to be armed.

- Program Cycle - read/write; this is a list of (position, settle time, dwell time). For example a sequence with a central ON and two OFF positions could be represented as:
  (0.0', 5, 20), (-5.0', 5, 20), (0.0', 5, 20), (+5.0', 5, 20)
  Here we assume, probably unrealistically, that it takes 5ms to move to each new position. The total cycle time for this sequence is 100ms. The position units are arc-minutes.
- Current position - read only
- Current state - read only
- Status

11.11.3 Commands

- Start program on the second timing event (must be sent 24-48ms in advance of TE) after the command is issued; runs until stopped.
- Stop on a selected program step
- Abort program immediately
- Validate a proposed program cycle to see if it is achievable (e.g., check against slew and acceleration limits)
- Stow, lock motors
- Reset status - clear the status bits
- Load standby program - loads the standby program into the active program
- Self-test

11.11.4 States

The nutator has the standard top-level states used for devices with precise timing requirements as described in section 11.6.3 above. The EXECUTING state, Figure 1, has the sub-states implemented in hardware shown in the following figure. A hardware monitor point is read to determine the nutator state.

The states are:

- MOVE - move nutator to position
- DWELL - wait at this position

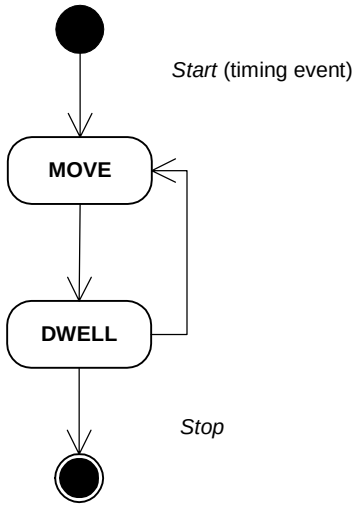
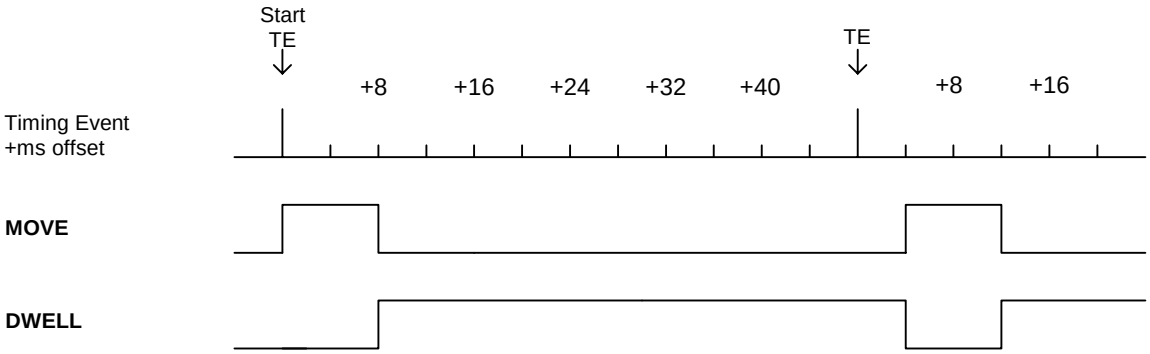


Figure 24 - Possible nutator hardware state diagram showing the sub-states of the Running state

11.11.5 Timing

As noted in the introduction, a program cycle only becomes active at the next timing event. Once nutating (running) the transitions between states occur on 1ms boundaries. The timing diagram, Figure 2, shows MOVE waits for the settle time from the time MOVE started. This means the settle time incorporates any time involved in performing the move.



We assume that none of the monitoring requests require timing more precise than that which is available from the CAN bus (~150 us), and hence they may be requested at any time.

Figure 25 - Hypothetical nutator timing diagram with programmed 8 ms settle times and 44 ms dwell times, started on the Start TE.

11.11.6 Class Diagram

Figure 3 is the nutator class diagram.

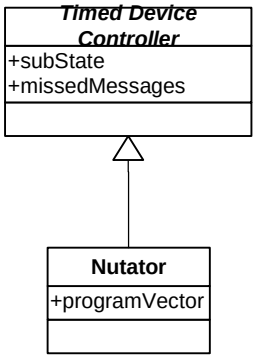


Figure 26 - Class diagram for the nutator.

The *programVector* is a logical ACS property sequence that is a list of triples containing the position, settle time, and dwell time.

11.12 Total Power

F. Stauffer
Last Changed: 2001-10-31

11.12.1 Description

Details related to the internal (embedded) software view of this device are not covered, and it does not cover the data production. *This device is currently under design so many details are still TBD.* The fundamental specification of the total power is that it produces data at 500 Hz for 6 total power detectors for each downconverter. Each antenna has one downconverter (single sideband) for the test interferometer and two downconverters (one for each sideband) for the final array.

When the total power device is enabled, the total power device collects and publishes into a data channel time-tagged data blocks aligned with timing events.

11.12.2 Parameters

Total power detectors - read only:

- Continuum Polarization 1
- Continuum Polarization 2
- Narrow Band1
- Narrow Band2
- Narrow Band3
- Narrow Band4

11.12.3 Commands

None available

11.12.4 States

The Total Power has the usual device controller top-level states as described in section 11.6.3 above. The ENABLED state has the following sub-states:

- COLLECTING - read total power data every timing event period and publish data.

Figure 27 - States for total power (inside the top-level ENABLED state)

11.12.5 Timing

The total power device does not have any time synchronized control behavior. Data for each timing event period are read in the 20 ms monitor point time window. The data produced in one timing event is 24 data values taken every 2 ms apart for each of six detectors or 144 - 16-bit values.

11.12.6 Class Diagram

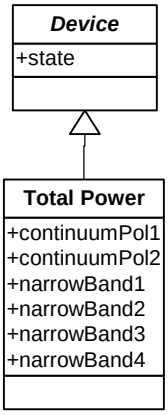


Figure 28 - Class diagram for total power device controller.

11.13Holography Receiver

M. Pokorny
Last Changed: 2000-10-13

11.13.1 Description

This section contains an overview of the control software view of the Holography Receiver device controller. It does not include the internal, embedded view of the device, nor does it include the analysis software view of the device.

The hardware design is currently underway, and many details of the design are TBD. It is expected that the holography receiver hardware will consist of two receivers, two mixers, oscillators, A/D converters, a digital signal processor, and supporting electronics (for example, power supply). The data group produced by the hardware consists of six values. These six values are produced from three analog signals (assuming that the quadrature signal is not produced digitally) that are converted by the A/D converters at a rate of 15 000 samples per second. The digital data is then processed and integrated to produce the six values in the data group. The integrations shall have a duration of 12ms, and shall begin 0, 12, 24 or 36 ms after a timing event. Four data groups are produced by the hardware every 48 ms. A continuous stream of data groups synchronized to the timing event is produced by the hardware.

11.13.2 Properties

Properties are labeled as read-only ("ro") or read-write ("rw").

- Reference frequency (rw)
- Signal attenuation (reflected signal or reference) (rw)
- Oscillator selection (ro)
- Gunn oscillator operation (on/off) (rw)
- Gunn diode PLL open/close (rw)
- Reset DSP (rw)
- IF signal level (reflected signal or reference) (ro)
- BB signal level (reflected signal or reference) (ro)
- LO level (ro)
- LO mixer current (ro)
- Heater current (ro)
- Tuning voltage (ro)
- Temperature (many) (ro): monitor a TBD number of temperature monitor points in the holography receiver enclosure.
- Data group (ro): a time-tagged list of twenty-four values that consists of the four holography data groups (of six values each) from the previous timing event interval.

11.13.3 ENABLED Sub-States

The holography device has the usual timed device controller top-level states as described in section 10.4. Although integrations begin on timing events, the device does not use time-synchronized control because all synchronization is handled in the hardware or firmware, and

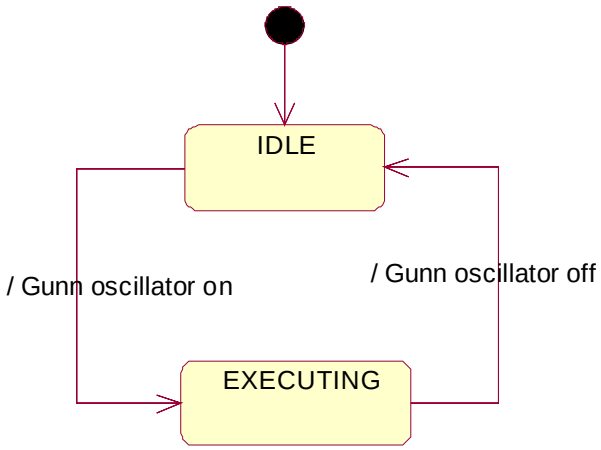


Figure 28- Sub-states of ENABLED; showing an example of a set of actions that may initiate state transitions.

the device control points themselves do not require precise timing.

•EXECUTING

The EXECUTING sub-state indicates that the data group monitor points are operational and contain recent holography data. Transitions to the EXECUTING sub-state may be be initiated by a command (for example, "power on Gunn oscillator"), or be "spontaneous" (for example, completion of "reset DSP" command). In the EXECUTING sub-state, the hardware will be producing holography data continually, and the software device controller will read the appropriate firmware monitor points in the timing window before every timing event (as described in ALMA computing memo #7) to retrieve the data. The data will be time-stamped by the device controller as it is read from the hardware device.

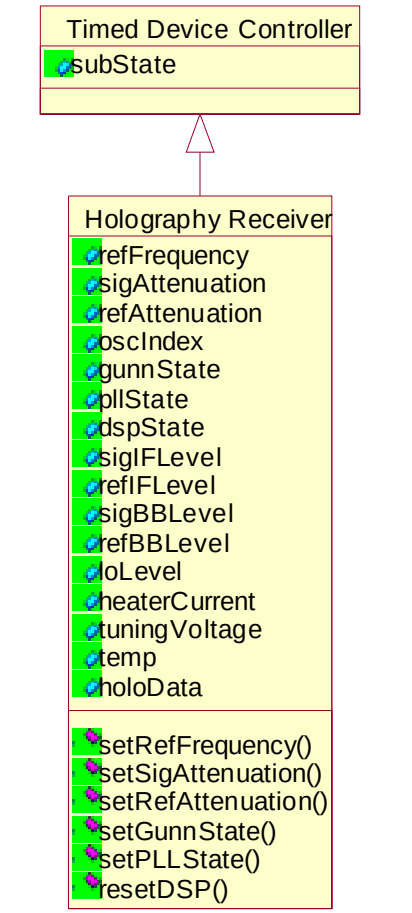
•IDLE

The IDLE sub-state indicates that the data group monitor points are not operational, and do not contain recent holography data. Various commands may initiate transitions from EXECUTING to IDLE (for example, "reset DSP" or "power off Gunn oscillator").

11.13.4 Timing

All control follows the standard ALMA hardware control timing specification. A holography data monitor request will be associated with the 48 ms interval ending on the timing event preceding the request.

11.13.5 Class Diagram



Class Diagram

12 Monitoring

12.1 Monitor point collecting

F. Stauffer
Last Changed: 2001-10-31

A monitor point collector will poll monitor data on the real-time systems and centrally concentrate the data on the ACC where it is distributed to applications including monitor point archiving and data production. The monitor collector is started on the real-time machines before the devices are instantiated. The monitor collector is a supplier to a notification event channel. The monitor collector polls the monitor points, and records the time tagged monitor data in an XML format chosen to be convenient for the monitor point archiving task (in particular, “rows” of data at a particular time are sent).

By default all properties are monitored at the rates defined by their characteristics. The monitor point collector has optional configuration that can over-ride or suppress these values.

The Properties are found at run time by querying the Manager for all local Devices, and then looking at the Properties in each Device.

Allowed characteristic monitoring rates are:

- 0.5 second
- 1 second
- 5 seconds
- 10 seconds
- 60 seconds
- 300 seconds

To synchronize monitor data across the array, polling is synchronized to the monitor rate. For example, a 1-second rate is polled on the 1-second mark, a 5-second rate on the 5-second mark, etc. The characteristic rates are not necessarily divisible by timing events (however, the system would be more consistent if the intervals were multiples of the 48ms timing period and this change should be considered). The monitoring uses the local time information described earlier in this document.

This implementation will be provided as part of TICS to be considered for integration with ACS at a later time. The present ACS "Logging and Archiving" implementation will not be used for this purpose (but it will still be used for log messages, for example).

12.2 Monitor Point Archiving

B. Glendenning
Last Changed: 2001-11-16

As described previously, blocks of monitor point values are collected by real-time computers attached to the hardware monitor points, and are then sent via a publish mechanism to a concentrator process at the center, which in turn publishes the blocks of monitor points to (potentially many) interested subscribers.

One subscriber is the monitor point archive task that subscribes to all producers of monitor point blocks. The archive task then pulls the blocks apart and places them into a RDBMS with an ODBC interface. Engineering data analysis is expected to largely take place through tools with an ODBC interface, although we may also have to write a few simple export tools (e.g., to comma separated value (CSV) text files).

No automatic action will be taken to purge the database – so the default is that monitor points will be accumulated forever. If it turns out that in fact it is desirable to purge the database from time to time, the database administration tools will be used to do this. (We expect the total monitor rate to be ~10kB/s = ~900MB/day = ~300GB/year). *(TBC - These values are considerable overestimates as they do not filter out the things faster than 0.5s that might dominate the total rate).*

There are three sets of RDBMS tables:

1. One set for both antennas.
2. One set for the central electronics.
3. One table for the meteorological data from the weather station.

The "sets" of tables for the antenna and central electronics consist of one table for each of the allowed monitoring rates in the system.

The first column of each table is the nominal sampling time, and is the primary key for the table. We require that the values in the row to have been sampled within ±10% (TBD) of the sampling interval around this time. For example, in the 0.5s table values must have been sampled within ±50ms (TBD) of the nominal sampling time.

There is then one column per monitor point, with the column name being the same as the name of the monitor point (which are guaranteed by the system to be unique). If a monitor point is not available at a particular time for any reason, its value should be *unset/null*.

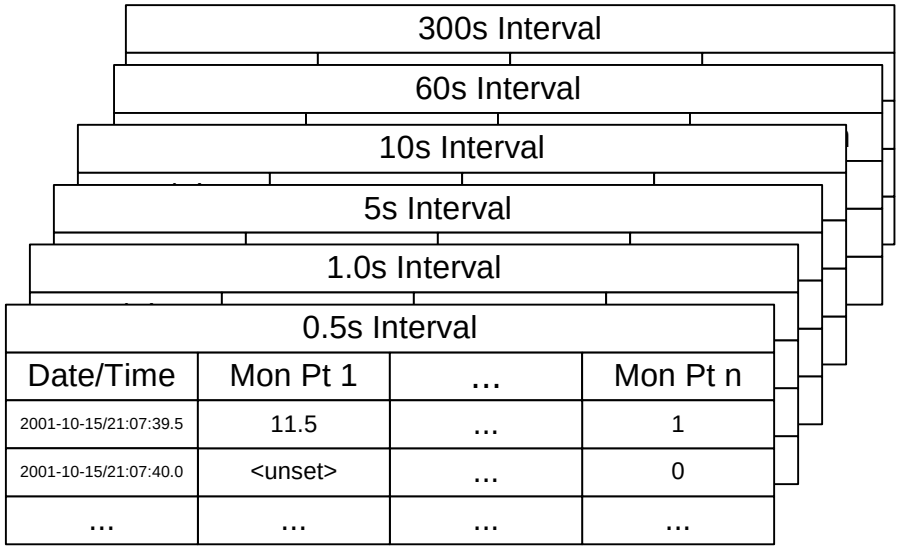


Figure 29 - A group of monitor archive tables.

Note that if the sampling rate of a monitor point changes, then it can appear in more than one table (its "old" rate and "current" rate).

Besides the pure monitor points, some TBD "quick look" (e.g., continuum channel phase) science data values will be archived and available for engineering and other purposes.

12.2.1 Issues

Do RDBMS columns have to be scalars? If so, we will have to have a naming convention for monitor points that are sequence). Other issues might involve limits on numbers of rows and columns.

There are concerns that this implementation may not scale well to large numbers of rows. If that turns out to be the case we will write utility programs that average the monitor data down considerably after some period of time, or alternatively we will consider alternate database schemas.

13 Model Servers

B. Glendenning
Last Changed: 2001-11-16

The term "model server" is used to refer to a well-encapsulated complex physical or astronomical calculation. The model servers are called by the ACC and results are passed on to the device or process requiring them. For example, the ACC calls the delay server to calculate the required delays that it then passes on to the CCC.

In general we would like model servers to have IDL interfaces so, e.g., they can be used from Java. As an expedient some of them may be implemented with only C/C++ API binding initially.

Since these servers will be called on the non-real-time ACC, their methods should generally accept a time (usually in the future) for which the calculation is desired.

13.1 Fundamental Astronomy Server

This software is used for items related to pointing and moving the antennas. The software underlying the interface is "SLALIB," created as part of the STARLINK project and distributed with ACS. The following capabilities are needed:

- Calendar conversions (*e.g.,* civil ↔ MJD)
- Timescales (*e.g.,* interconvert UTC, LAST, TAI)
- Precession, nutation, aberration, refraction (the radio refraction model will be provided by ALMA), proper motion
- Celestial coordinate conversions, *e.g.,* FK4/FK5, galactic, AZ & EL
- Terrestrial coordinate systems
- Ephemeris calculations for solar system objects

13.2 Delay Server

The delay server is principally used to calculate delays and delay rates for the correlator. The underlying software is based on the CALC software from the Goddard Space Flight Center, VLBI group.

Item	Description	Source
Time	Compute delay as a function of this time	
Station information (x2)	Frame information (geocentric?), Each station x,y,z (m), Non-intersecting axis offset (m), station name, axis type (alt-az)	Configuration file
Source info	RA, DEC (rad) and proper motions (arc-sec/year), reference date, parallax, source name (not needed?)	Source catalog or observing script.
Earth orientation	EOP epoch date, TAI-UTC, UT1-UTC, earth pole offsets	USNO data in local table plus interpolation calculations. Must have

		table update procedures.
Weather	Surface pressure (millibars?)	Weather station monitor points.

Table 13-1 – Principal inputs for the delay server

Item	Description
Time	The time (UTC?) for which the calculated results have been determined.
Delays and Rates	Calculated for: the group delay, the dry atmosphere, and the wet atmosphere. Units are sec for delays, and sec/sec for rates. *TBD: can the rates be used for the fringe generator?*
Pointing position and rates.	Azimuth, Elevation (rad), and Azimuth, Elevation rates (rad/sec) for the second station of the baseline. This might not be used but should be useful for debugging.
UVW	UVW coordinates for baseline (m) in J2000.0 frame.

Table 13-2- Principal outputs of the delay server

14 Data Production

B. Glendenning
Last Changed: 2001-10-05

14.1 Overview

Data production refers to the process of producing data files that can be used by reduction packages to derive calibration values or to be used for astronomical data reduction.

For the test interferometer, optical telescope observations will use a TBD format compatible with the TPOINT program written by P. Wallace that will be used for pointing reductions. All other observing modes will use a FITS based format unique to the ALMA test interferometer that has been chosen to be close to the data structures of the IRAM Gildas package that will perform the calibrations. This format is described by Lucas and Glendenning (2001 – TODO insert ref). Gildas in turn can export data to other formats (e.g., UVFITS) for those who wish to use other packages.

14.2 Data Distribution

In general, Data Production requires access to both "backend" data and monitor data. As described previously, both monitor and backend data are collected, time-tagged, and buffered at the real-time computer attached to the particular Device that then sends the data buffers to a central process that in-turn redistributes the data to processes that have need for the data.

While observing, there will be either one or two (when observing "independently" with two single dishes) Data Production tasks that subscribe to the monitor data and appropriate backend data streams.

The data will be distributed Data using the Data Channel mechanisms described in a previous section.

For the test interferometer, it is assumed that an observation is completed before starting another. This means overlapping observations are not supported.

14.3 Processing

Data production is entirely data driven. That is, once configured, it proceeds solely by examining the data that is sent to it. It does not have to interrogate other software subsystems. A new file is started for each new observing session (loosely: a series of observations contiguous in time intended for a single purpose).

Data processing follows the following sequence:

- Data is first buffered in the Data Processing task for a sufficient length of time to overcome the latency in the system (that is, to assemble all data necessary to handle an integration). Monitor data is retained so that values are available from before and after a particular integration
- Further processing is skipped - no data is written - if the antenna is still coming onto source or the frequency has never been locked for the current integration.
- Flags are calculated. Flags are calculated conservatively (*i.e.,* data is flagged bad if the bad occurrence might have happened any time during the observation). Data flagging parameters will include:
    - Antenna on source
    - Nutator in position
    - Frequency locked (CRG, 1st and 2nd LO's, Frequency reference)
    - Roundtrip phase correction
    - Total power level
- A few critical monitor points (in particular, the mount position) are interpolated to the time of the backend data.
- Other derived values (*e.g.,* UVW) are calculated.
- Some "quick look" (e.g. phase) information is calculated for external processes that might be interested in it.
- Single dish data that has several "phases" (e.g., ON/OFF) is (optionally) further integrated in the data production task.
- When an integration is completed, data rows are formatted, written to disk, and flushed. Data is written on either a per-integration, or per block of integrations (for efficiency) basis (TBD).
    - Provision is made for two integration times. A short continuum time corresponding to the average of a specified channel range, and a much longer

spectral integration time. These will result in separate FITS header data units in the same file.

- All monitor data is resampled as necessary to be coincident with the integrations at the longer integration period.

- Tasks that have registered as being interested in the data are notified that more data is available.

Data production continues until *stopped* (at the end of the current integration) or *aborted* (immediately).

14.4 Observing Mode Data Production Details

14.4.1 Optical Telescope Data

A TPOINT format. Details TBD.

14.4.2 Radio Data

The data format for all radio observing modes is specified by Lucas and Glendenning (2001). Its important values include the following.

What	Source
Primary data	Test correlator (cross *or* auto correlations) or holography receiver
Scan & observation number.	Calculation (count). 1-relative
Observing time	Master clock and calculation (e.g., LST)
Observing mode	Observing script
Project ID	Observing script
Az/El	Antenna encoder monitor values
Position on sky	Calculation
Site information (e.g., longitude and latitude)	Configuration database
UVW	Calculation(source information, site information, time)
Total power	Total power detectors
Weather information	Weather station (via monitor points)
Flags	Determined from monitor points
Aperture and beam efficiency	Calculation
Load temperatures	Configuration database?
Receiver and system temperatures	Calculation

Focus location	Monitor point
Source parameters (e.g., position, flux, Doppler parameters)	Observing script
Array geometry	Configuration database
Frequency	Observing script and monitor points
Monitor data	All monitor data sampled during the observation period will be included in the file. It will not be resampled to correspond to the observation times.
Side band gain ration calibration	Spectroscopic observation of line of known intensity in upper or lower sideband.
Forward efficiency	Skydip in total power.
Aperture efficiency (Jy/K conversion)	Measurement of antenna temperature on point source of known flux.
Axes offsets	Baseline measurement (fitting the residual elevation dependence of phase).

15 Logging, Errors, Alarms

F. Stauffer
Last Changed: 2001-05-05

15.1 Logging

Logging is part of the ACS. Logging collects information from the distributed systems and centrally concentrates the logging where it is distributed to applications such as archiving and user feedback. Logging is built on the CORBA notification service.

ACS logging is based on CORBA Telecom Logging Service which in turn is based on the CORBA Notification Service. Each log is a record with a time stamp, info on the object sending the log, its location in the system, and a formatted log message. Each computer has a logging manager that is a notification supplier writing sequences of structured messages to a notification event channel. Logs are collected on a central host by a notification consumer. The logs are based on XML and are made available to applications. TICS will provide a log message archive application using IBM's DB2 database.

ACS will provide a logger class interface that applications use to log messages. Log messages have a type and a severity, and log messages can be suppressed at the source. ACS will provide a JAVA based log message browser with the features listed in section 8.3.

15.2 Errors

The error system is part of ACS. Software commands that fail generate errors. When a user request fails, an error stack is created to record the error, and the error stack is passed back up the call chain until the error is handled or logged. Each level of return or exception can either handle the error, or add information to the error stack, or log the stack and delete it.

Error logs need to prevent flooding the error log. An error state flag can be used. When an error happens, the error state is set and logged, and when the error is removed, the error state is cleared and logged.

15.3 Alarms

The alarm system is part of ACS. Alarms are produced by properties that are outside of normal operation. Properties can trigger alarms on values outside of range, change on value, etc. Alarms go into a Notification Service channel where they can be subscribed to by applications.
Alarms are recalculated when a Property changes. When an alarm is triggered, the alarm is written to the alarm notification channel. Alarms are distributed locally and are sent to the ACC where they are logged and distributed. Applications requiring the alarms are attached as consumers to the alarm notification channel.

16 Relation with other ALMA Software Activities

16.1 Science Software Requirements

G. Harris
Last Changed: 2001-02-07

Specifications and use cases from the SSR committee generally apply at an observatory management level and do not influence the test instrument.

But use cases that cover the technical aspects of observing modes, calibrations, and other actions performed on the instrument do apply to the Test Interferometer. Some of the use cases for the final ALMA array have been edited appropriately for the test interferometer and are available in Brooks *et. al.* (2001).

16.2 Analysis and Design

G. Harris
Last Changed: 2001-02-07

Like documents from the SSR committee, the High Level Analysis [HLA] document applies to the final ALMA instrument, and not the test instrument. It covers the identification of major packages and functions of the final instrument for observing, planning, observatory management, science goals, etc.

Some concepts from the HLA document may be used in the TI as they become available. One goal here is to minimize the path from test to final instrument.

16.3 Telescope Calibration

B. Glendenning
Last Changed: 2001-05-05

Telescope calibration data reduction is being implemented in the IRAM Gildas package, except for telescope pointing which is also available with the TPOINT software written by P.T. Wallace (and will be used solely for optical pointing data). The data format for radio data is a FITS based format described by Lucas and Glendenning (2001). The data format for TPOINT will probably be "Observation record format #4", in which the raw and observed az and el are recorded.

The control software interaction with the radio calibration package proceeds as follows:

- When the current Observation completes the several FITS Header Data Units (HDU – TODO – insert acronym) will be written. The data includes both "backend" and monitor data.
- In a text file nominated at startup TICS writes the name of the FITS file, the number of the last HDU, and the time at which the Observation completed.
- The external calibration package polls the text file, and when it has changed looks at the FITS files for the new data and then performs its calculations.
- If any calibration values are required to by TICS, the calibration package writes a Python script that inserts them into the TICS system. The name of the Python script contains the time of the Observation from which it has been derived.
- If the calibration should be applied immediately, the control software is notified by changing a symbolic link (for each type of calibration) to point to the new link. (That is, TICS polls the symbolic links waiting for them to change). The new script is executed, replacing the current calibration values with the ones in the script.

The following calibrations will be implemented. The "calibration parameters" column gives the calibration parameters that in turn will be used by the control software (e.g., it includes pointing coefficients but excludes panel offsets calculated during holography).

Calibration	**Calibration Parameters**
Optical pointing Determine the primary pointing model of the antenna through the observation of a large number (~100) of stars with an optical telescope.	Pointing coefficients (~11) (per antenna)
Single dish pointing Five point observations or equivalent on a bright point source.	Pointing offsets (per antenna, per band)
Interferometric pointing Interferometric five-point observations or equivalent on a bright point source.	Pointing offsets and sag (per antenna, per band)

Temperature scale For the test interferometer, probably observe a switching load in the subreflector (TBD)	T_{Sys} (per antenna, per band, per pol'n)
Flux For absolute flux calibration. Typically observing a planet.	Gain (per antenna per band)
Delay Observe ~100 bright sources.	Delay offset
Bandpass Observe a bright continuum source. (Or: Photonically?)	Gain (per antenna, per band, per baseband, per pol'n, per spectral point)
Baseline Interferometrically observe a large number (~100) of bright sources.	X,Y,Z offsets (per antenna, per band (?))
Phase Astronomical phase calibration on a point source near (~1°) the primary source.	Phase (per antenna, per band, per baseband)
Polarization Observe a bright source of known polarization over a wide range of parallactic angle.	Gain (per antenna, per band)
Holography Single dish: Raster scan a beacon with the holography receiver. Interferometric: Raster scan a SiO maser with one antenna with the other fixed.	Nothing for the online system

17 References

Brooks, Mick, and D'Addario, Larry 2001. ALMA Monitor and Control Bus Interface Specification, ALMA08001.001/ALMA-SW-0007.

Brooks, Mick, *et. al.* 2001. Test Interferometer Control System Requirements Summary. Computing Memo #TBD.

Chiozzi, *et. al*, 2001. ALMA Common Software Architecture. Exact reference TBD.

D'Addario, Larry R., 2000. Timing and Synchronization, ALMA Memo #298.

Lucas, R., and Glendenning, B.E. 2001. Test Interferometer Data Format. Exact Reference TBD.

Pisano, Jim, 2001. ALMA Test Correlator Control Computer Software Design. Computing Memo #TBD.

Raffi, G., and Glendenning, B., 2000. ALMA Common Software Technical Requirements. ALMA-SW-0005.