Atacama
Large
Millimeter
Array

Device Driver Code Generation Framework

COMP-70.35.05.00-003-A-DSN
Version: A
Status: Draft
2010-06-08

Keywords: Software Design, Monitor and Control, Code Generation, Device Driver

Allen Farris Thomas Juerges	NRAO NRAO	2007-05-01 2009-02-10

Change Record

Version	Date	Comment
1.0	2006-12-12	Initial version.
A	2007-05-01	Updated as a result of comments and enhancements. Also changed formatting for compatibility with AEDM standards.
A	2009-02-10	Added an explicit guideline that Microsoft Excel is the mandatory editor.
A	2009-05-18	Added Ethernet Device code generation
A	2009-08-11	Corrected the documentation about control points and removed enumerations as world types for Monitor points.
A	2010-06-08	Disallow default values for sequences. Removed ubyte from allowed RAW data types.

Table of Contents

1. Introduction

In this discussion it is assumed that the reader has a general knowledge of a hardware device and that a hardware device is described by a document called an ICD (Interface Control Document). It is also assumed that one is aware that devices are accessed via a bus (a CAN bus or an Ethernet bus in the case of ALMA) and that these devices are commanded by placing structured messages on this bus. The structure of these messages includes a node address that is used to route the message to the proper device (CAN-bus) or use a point-to-point connection (Ethernet-bus) and a relative address that is used to identify a specific command to that device.

Monitor and control points are the mechanism by which software monitors and commands hardware.

Within the ALMA context, a monitor or control point is a single command sent to a specific address of a node on the CAN bus or to an Ethernet-device that is connected via TCP or UDP which causes some action to occur in the device. In addition to the specific address, all commands have a name that is unique to that device.

Commands cause one of two types of actions to occur: either they cause the hardware device to send a certain pattern of bits to the requestor on the bus (which is not supposed to modify the internal state of the device), or they use the contents of the command to modify the internal state of the device (which may, in turn, cause the device to perform some action, such as set a frequency). Actions of the first type are called monitor points; the second are called control points.

The data associated with a request, either the data returned by the device or the data sent to the device, have, in the case of CAN-bus devices, two aspects: a “raw” data type that is understood by the firmware and hardware internal to the device, and a “world” data type that is understood and read by humans. The world data type frequently corresponds to some quantity in the physical world such as a voltage or frequency and is the only one that Ethernet-devices understand in ALMA. Units are almost always associated with this world data type. The raw data type, by contrast, is merely a sequence of bits, which may represent integer or floating point numbers or merely be bits. The conversion for CAN-bus devices between the world data type and the raw data type is specified in the ICD and the software is expected to perform this conversion.

Data associated with monitor and control points are constrained in various ways that are dictated by the structure of the device. For example, voltages may be constrained to certain limits by the structure of the hardware; only certain bits in a status word may have meaning. If invalid data is sent to a control point, it is usually ignored. If invalid data is returned by a monitor point it usually indicates a serious problem internal to the device. The specification of constraints on monitor and control points is an essential part of the ICD that has a major impact on the software.

So far we have identified the following aspects of monitor and control points.

- Name
- Address
- Raw data type (CAN only)
- World data type (CAN only) or Data Type (Ethernet only)
- Specifications on how to convert between raw and world data (CAN only)
- Constraints on the data

Actually, the name of the command may be thought of as the world name of the command whereas its address is the raw name. These aspects are essential properties of all monitor and control points. They may be thought of as aspects of the commands themselves and their "internal" characteristics, so to speak.

There are many other aspects of monitor and control points, but these have a different focus. They specify "external" aspects of the commands: e.g. how the command relates to the lifecycle of the device, or to time, or when it is appropriate to use the command. We now turn to these additional aspects of monitor and control points.

Monitor and control points may be used in one or more of three modes:

- Startup
- Diagnostic
- Operational

Startup is the time from which the device is first powered up until it is ready to operate. Diagnostic mode is intended for detailed testing, usually interactive, and under close scrutiny by an experienced engineer. Operational mode is the "normal" mode of operation, i.e. automated and under the supervision of software. Any command may be used in a diagnostic mode. However, commands designated as "diagnostic" may only be used in a diagnostic mode. Commands designated as "startup" are intended to be used only during the startup interval and are not normally needed during operational mode. After the startup interval, only commands designated as "operational" may be used in an operational mode; diagnostic commands may not be used in an operational mode (except under unusual circumstances). There is not normally a shutdown mode, because most devices do not require special handling during shutdown. However, if there were such considerations, it would be necessary to add shutdown as another mode.

This classification of commands into startup, diagnostic and operational is really a property of the device as a whole; the classification relates commands to the lifecycle of the device. Another aspect of lifecycle considerations is to further characterize the startup interval. Startup is the interval of time between when the device is first powered up to the time the device is capable of giving valid readings and performing valid actions. This time can vary considerably; it may be milliseconds, many seconds, or, in some cases, minutes.

An extremely important aspect of monitor and control points is how they relate to timing events. Commands that must be executed on precise timing boundaries must be specifically designated and require specialized handling by the software. This aspect of monitor and control points is not valid for Ethernet-devices due to the lack of timing determinism in the TCP and UDP protocols.

There is another aspect of commands that is more difficult to characterize but relates to how the command is intended to be used. This specification may take several forms. For example, an ICD may specify how frequently a monitor point should be sampled, or it may specify that a particular command is intended to be used after the detection of a fault in order to obtain additional information about the fault. In some cases an ICD may specify that a monitor point be sampled at a high rate, e.g. every 48 milliseconds, looking for specific anomalous conditions.

The final aspect of monitor and control points we will mention is priority. Monitor and control points may have a priority associated with them. This is especially relevant to commands that require specific timing and is used to resolve conflicts when multiple commands must be executed within the same timing window. The current Control software does not directly implement a concept of priority. Actually there is an implicit priority concept in that commands are divided into two categories: ones that must be implemented during the next timing event window and ones that are implemented on a first-come first-served basis as time allows (again CAN-bus devices only). This technique has proved to be adequate so far.

As a summary we have the following aspects of monitor and control points.

- Name
- Address
- Raw data type (CAN only)
- World data type (CAN only) or Data Type (Ethernet only)
- Specifications on how to convert between raw and world data (CAN only)
- Constraints on the data
- Mode of use: startup, diagnostic, operational
- Relation to timing events
- Intended use

It might be useful to give a specific example of a monitor point from a real CAN-bus device. The following is from the Fiber Optic Amplifier Demux (FOAD).

- *Name*: GET_PSU_AMP
- *Description*: Measured value of the EDFA supply current. Values are produced using a 10-bit analog-to-digital converter; therefore valid readings will always be between 0 and 1023. An invalid result can be returned (see Validity of Readings). This reading is not meaningful when the EDFA PSU is shut down.
- *Address*: 0x0002a
- *Raw data type*: int16
- *World data type*: float, Amps
- *Conversion from raw data to Amps*: Amps = 0.00474609375 * Raw value
- *Constraints*: max value = 5.0 min value = 0.0

- *Mode of use*: operational
- *Related to Timing event*: no
- *Intended use*: sampled every 10 seconds
- *Priority*: none

2. The Device Driver Code Generation Framework

Note: A version of this section has been submitted for publication in the proceedings of the ADASS XVI conference held in Tucson, October 15 – 18, 2006.

2.1. The Problem

A control system for a modern radio telescope has a large number of types of hardware devices that must be monitored and controlled by software modules. The ALMA telescope will have more than 50 types of devices and between 3,000 and 4,000 instances of those devices. To make matters worse, during development, hardware is sometimes redesigned, resulting in prototype and production versions with substantially different properties. Writing software to support these hardware devices is challenging.

The ALMA telescope control system uses a model-driven approach to solve this problem. Information from the hardware ICD (Interface Control Document) is captured in the form of spreadsheets that contain detailed information about the device, its monitor and control points, and its properties that are to be permanently archived. The spreadsheets are saved as XML documents that are used within a model-driven process to automatically generate software products to support these hardware devices.

This technique results in a dramatic improvement in productivity, enabling the software developer to concentrate on higher-level aspects of devices. Especially during a development phase, it also provides the ability to accommodate changes resulting from hardware redesign far easier than in the past.

2.2. Device-Level Software and Model Driven Software Development

A hardware device is described by a formal document, called an Interface Control Document, jointly developed by and agreed to by both hardware and software development groups. An ICD specifies precisely how a hardware device interacts with the external world; both monitor and control points are described in detail. A software module must be constructed that provides for: basic control of the device, fault detection, reporting, and recovery, and storing monitored properties in a permanent archive. Since ALMA has about 50 types of devices to support, assuming 25 monitor and control points for each type of device means implementing and testing 1,250 methods.

The really old-fashioned way to solve this problem is to start programming: 1 down and 1,249 to go; or, outsource the problem to an army of programmers. A better approach is to use the well-known object-oriented approach to generalization: inheritance, virtual functions, templates to capture common features combined with a table driven approach. However, this approach can frequently result in complex code.

An alternative approach is to develop a model of the problem and use a code generation framework. The complexity inherent in the problem is embedded in the model – not in the generated code. The generated code is simple. The ALMA control system uses this latter approach. Information about the hardware devices from the ICD is captured in the form of spreadsheets that are saved as an XML document. This XML file becomes the input to the code generation framework.

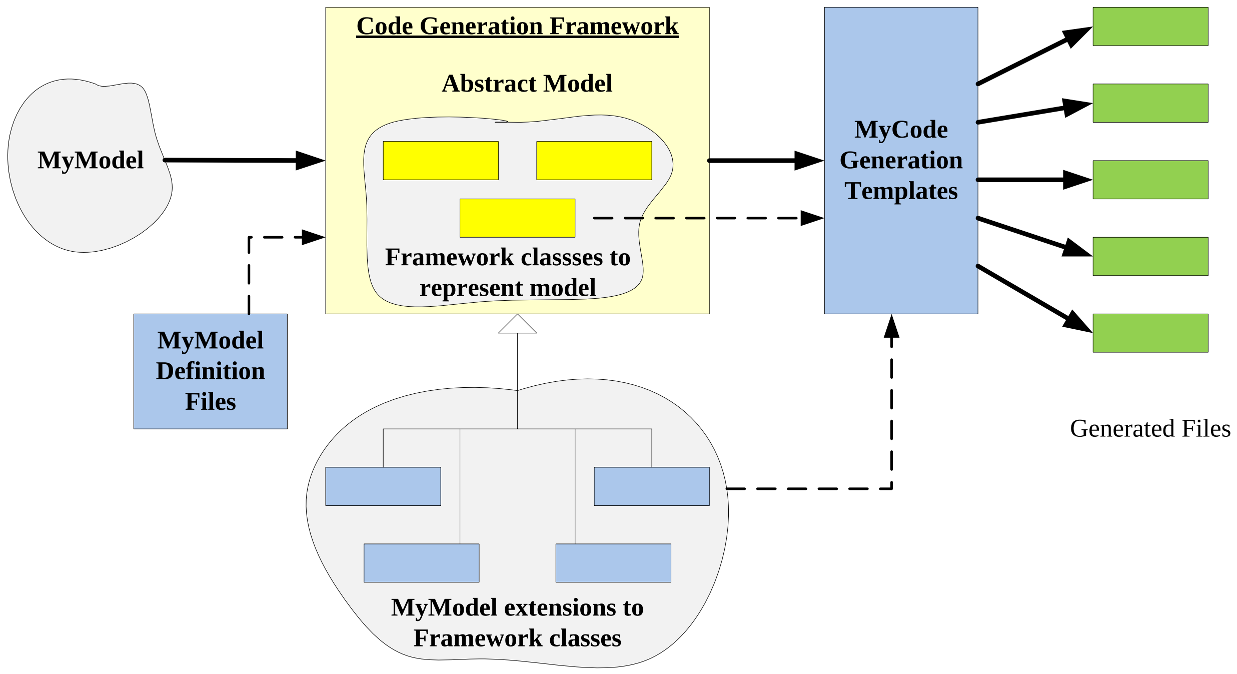
Code Generation Framework
Abstract Model
MyModel
Framework classses to represent model
MyModel Definition Files
MyCode Generation Templates
Generated Files
MyModel extensions to Framework classes

Figure 1: Model Driven Software Framework

The ALMA Control system uses the software framework from an open source software project called openArchitectureWare, found at openArchitectureWare.org. Its basic structure is described in Figure 1. The framework allows one to define a model, which is an abstract concept that can take many forms, such as a UML diagram or an XML structure. Definition files describe your particular type of model to the framework. Java classes internal to the framework allow one to access the data contained in the model. These are usually extended to support your particular type of model.

The code generation framework instantiates an instance of your model and uses template files to generate output files. These template files are written in a very readable markup language that accesses the classes containing data from the instantiated model. Any output code that is generated is as readable as you make it, because you provide the code generation templates.

2.3. Device-Driver Spreadsheets and Code Generation

Four named spreadsheets form the input to the code generation process: Main, Monitor, Control, and Archive. The structure of these spreadsheets is based on a detailed analysis of the actual contents of ALMA hardware ICDs. The Main spreadsheet contains the name and description of the device and its assemblies, the specific name and version of the ICD. The ALMA control system uses the CAN bus and Ethernet, so this spreadsheet also contains data needed to access either the device on the CAN bus or connected via Ethernet: its CAN bus number, relative CAN address and a base address. In the case of an Ethernet-device, it does not contain any CAN-bus related information but the name of the “Vendor Class”. See chapter 4 for a detailed description of the differences between CAN-bus devices and Ethernet-devices.
The Monitor spreadsheet for CAN-devices contains the relative CAN address of each monitor point, its raw data type (such as a 16-bit signed integer), its external data type and units (such as a float in volts), how to convert between raw data and external data, validity ranges, error conditions and actions to take if these are triggered, as well as operating modes (startup, operational, diagnostic) and descriptive text. If Ethernet-device code is generated, the Monitor page must not contain CAN-related information, nor can it contain any conversion information between raw and external data. All data coming from or going to Ethernet-devices is already “ready to use”.
The Control spreadsheet is similar to the Monitor spreadsheet but adds descriptions of input parameters.Ethernet-devices do not require special descriptions of the input parameters since all input values are already in a proper format.
The Archive spreadsheet specifies how frequently properties are to be sampled, how these are related to monitor and control points, and data needed to display them graphically.

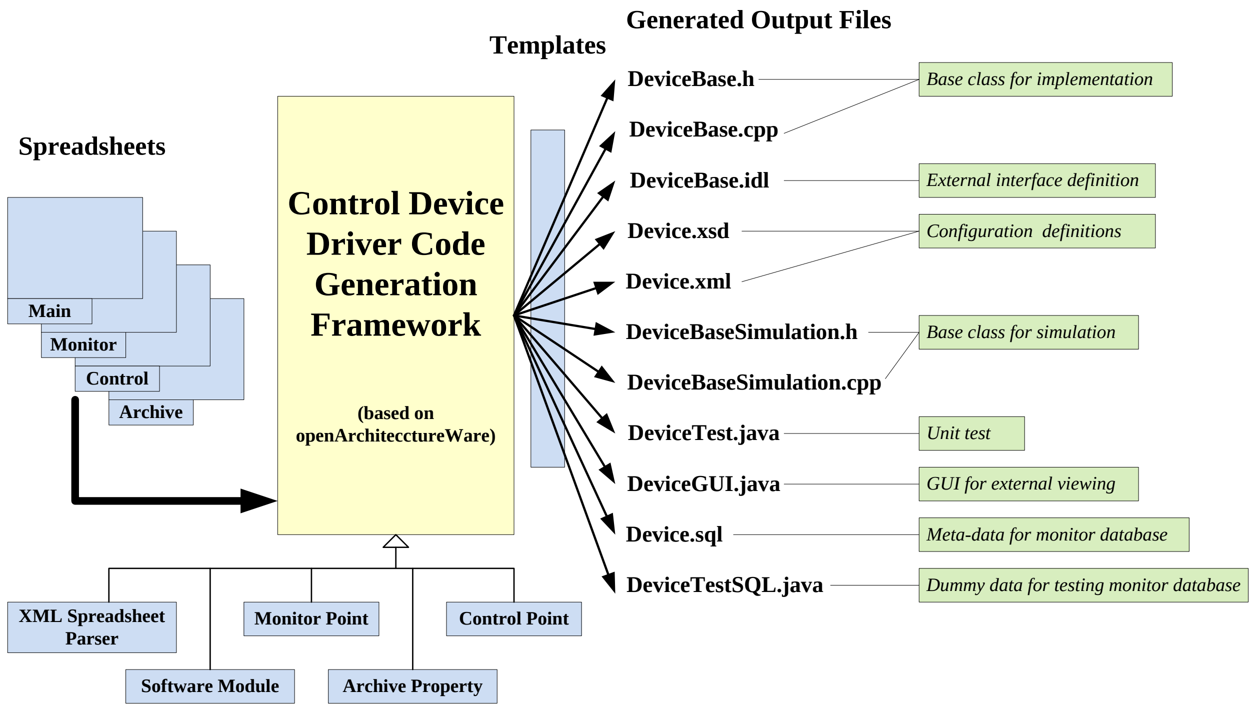

Figure 2: Generating Software Products for CAN-bus based devices

Spreadsheets must be edited in Microsoft Excel and have to be saved in Microsoft Excel XML-format in order to preserve compatibility, readability and to keep them editable for other developers. A simple parser translates the XML document into a three-dimensional array of strings: each cell is indexed by worksheet number, row and column. This array of strings drives the code generation process, depicted inFigure 2. A variety of output products are produced. First, base classes in C++ are generated to support low-level access to all monitor and control points. These are extended by the software developer to add higher-level functionality. CORBA IDL definitions are generated to allow access to the device from the distributed network. Configuration files are generated as are base classes that drive a device-level simulation. Units test are also generated. Meta-data that describes the device and its properties to the monitor database are generated as well as data to test its interface to that database. Finally, GUIs can also be generated that display that data.

2.4. Characteristics of the Generated Software

Since the output products are generated from a common framework, consistency is achieved across all devices in implementing various design policies. For example, the generated code enforces a state model (start, configure, initialize, operational, error, and stop) with well-defined transitions between these states. This is especially important to error handling. In each software device-driver faults are detected and reported to its parent module and action, as specified in the spreadsheet, is taken. If the action is highly specialized it is defined as an abstract method that must be implemented in the extended classes.

A common practice in hardware design is to package a number of status bits into a byte sequence that is reported in a single command. Such a sequence of bits is defined in the spreadsheet and the generated code unpacks these bits, making each one a distinct property. Frequently faults are tied to specific values of these status bits.

Each software device-driver has a monitor thread that samples properties of the device at varying rates and stores those properties in the monitor database. This monitoring thread can be turned on or off by the software, depending on the state of the device. In addition to this routine monitoring, the generated code also inserts records into the monitor database to denote state changes and faults. In addition, when certain specified data are changed via control points, similar time-stamped records are stored in the monitor database. In this manner, a complete life history for each device is contained in the monitor database: when it started and stopped, when crucial data values were changed, when faults occurred, as well as the usual continuous sequence of sampled data points.

One way to view this model driven approach is to recognize it as an alternative method for implementing generalizations in software development. Methods for dealing with generalization in object-oriented software development are well known. The lesson learned here is that certain kinds of generalizations are much easier to deal with in a rich model-driven code generation framework.

3. Spreadsheet Structure and Values of Cells in the Worksheets

There are four worksheets that characterize the hardware device, named: Main, Monitor, Control, and Archive. The first row of each spreadsheet contains a single cell that is the name of the spreadsheet. The names in the first row are:

- Hardware Device
- Monitor Point
- Control Point
- Archive Property

As an illustration, a completed spreadsheet for the Holography receiver, accompanies this document. See **Holography_spreadsheet.xls**.

With the exception of the Main spreadsheet, and the first row of each spreadsheet, each spreadsheet contains a fixed number of columns.

The following sections list the possible values for the columns of the worksheets. All cells **MUST** be filled with a value. A blank cell will break the parser (likely with a weird error message) and will probably result in an invalid spreadsheet. Likewise, there should be no blank rows at the end of a spreadsheet.

The following is a list of valid types of values that can populate a cell in the spreadsheets.

- **Name** *A string of characters from the set {'a'-'z', 'A'-'Z', '0'-'9', '_'} that begins with a letter {'a'-'z', 'A'-'Z'}. Names are limited to 32 characters. Names cannot have embedded spaces. No Unicode characters!*
- **Text** *Any sequence of ISO-8859-1 characters with ASCII codes 0 - 255.* **Note:** Do not insert any Unicode, i.e. UTF-8 or UTF-16 characters!!! *(This may be relaxed in the future to allow characters outside the strict ISO-8859-1 limit.)*
- **Number** *A number in normal decimal format, e.g. 42 or 3.14159.*
- **Hex** *A number in hexadecimal format, e.g. 0x31, capitalization of letters is not important.*

In the list of valid values for cells in the worksheet that follows, if a string other than the above appears, it is assumed to be a literal. In some cases these types are combined with other literal characters; for example the notation "^Name" means the character '^' followed by a valid Name.

When a spreadsheet is processed by the code generation framework, there is a detailed validation step that checks for adherence to the rules laid out in the following sections. The error messages are intended to make the job of detecting and correcting errors in the spreadsheet fairly easy. The validation step is not 100% effective, but if the spreadsheet passes the validation step, the code generation process will usually generate code without error.

3.1. Main

The column names in the Main spreadsheet must be:

- Assembly
- Description
- Device Name
- Vendor Class (Ethernet only)
- Extends
- Parent (CAN only)
- Cardinality (CAN only)
- Node Address (CAN only)
- Channel (CAN only)
- Base Address (CAN only)
- ICD
- ICD Date

The first row of the Main spreadsheet must contain the name "Hardware Device". The second row contains the names of the columns. The third describes the hardware module as a whole and must contain a value for each column. The first column is the name of the hardware device, i.e. the Line Replaceable Unit (LRU) – in the case of the holography receiver, "Holography". The holography receiver has two assemblies: HOLORX and HOLODSP. These are entered as the fourth and fifth rows. For these rows, the 'ICD' and 'ICD Date' columns are not filled in, since they are already present in row 3. In the case of a hardware module that only has one assembly only the third row is filled in; in this case the name of the LRU and the assembly are the same.

Following the third row describing the LRU, and additional rows describing assemblies if any, one can list an arbitrary number of rows containing notes. The name in the 'Assembly' column must be "Note" and the notation is placed in the 'Description' column. This is the only place one may insert notes.

3.1.1. Assembly

Valid values

- **Name**

Discussion

This cell should contain the acronym by which the main device is known. This should be a short name, e.g. LORR. This name is used in the code generation framework to identify the software module that corresponds to the hardware device identified by the ICD.

3.1.2. Description

Valid values

- **Text**

Discussion

This cell should contain a brief description of the main hardware device.

3.1.3. Device Name

Valid values

- **Text**

Discussion

This cell should contain the full name by which the main device is known, e.g. Local Oscillator Reference Receiver.

3.1.4. Vendor Class (Ethernet only)

Valid values

- **Text**

Discussion

This cell should contain the full name of the C++-class that implements the Ethernet communication layer. It refers to the names of the automatically included header files and the implementation file, too.

3.1.5. Extends

Valid values

- **none**
- **Name**

Discussion

This code generation framework supports inheritance and this cell is used to identify a previously defined module from which this module is extended. The “Name” is the assembly name of the previously defined module. If this device is not an extension of a previously defined device, “none” should be entered.

3.1.6. Parent (CAN only)

Valid values

- **root**
- **Name**

Discussion

The code generation framework supports a part-whole model. This cell identifies the “whole” in the part-whole model. A hardware module may consist of more than one assembly. The entire hardware module is referred to as an “LRU” – Line Replaceable Unit. The hardware module as a whole always has “root” in this parent cell. There is always at least one assembly, but there may be more than one. For example the Holography receiver contains two assemblies: HOLORX (the holography receiver) and HOLODSP (the holography data products). The main hardware module is called “Holography” and its parent cell contains “root”. The HOLORX and HOLODSP rows both contain “Holography” as identifying the parent of these assemblies.

3.1.7. Cardinality (CAN only)

Valid values

- **none**
- **1**
- **N**
- **0..1**
- **0..***
- **1..***
- **0..N**
- **1..N**

Discussion

If this spreadsheet identifies a part-whole model, this cell identifies the cardinality of the part in relation to the whole. “1” means exactly one part; “N” means exactly N parts; “0..1” means optional; “0..*” and “1..*” mean either 0 or 1 to many parts; and “0..N” and “1..N” mean either 0 or 1 to a specific number of parts. "N" must be an integer. If this spreadsheet does not identify a part-whole model "none" should be entered.

3.1.8. Node Address (CAN only)

Valid values

- **none**
- **parm**
- **Hex**

Discussion

This cell should contain the default node address on the CAN bus at which the hardware device is located. In some cases there is no node address, as in the case of Holography, which is a logical concept of a container; the actual node addresses are associated with its assemblies. In such a case "none" is entered. In other cases, there is no default node address, as in the case of multiple instances of the hardware device; in this case the node address must be entered as a parameter and "parm" is entered.

3.1.9. Channel (CAN only)

Valid values

- **none**
- **parm**
- **Number**

Discussion

This cell is similar to the node address; it is the number of the CAN bus on which the device is located. Likewise, if there is no channel, "none" is entered. If there are multiple devices and the number must be entered as a parameter, "parm" is entered.

3.1.10. Base Address (CAN only)

Valid values

- **none**
- **parm**
- **Hex**

Discussion

In some cases there are multiple devices at the same node on the CAN bus and the devices are distinguished by separate address spaces within the same range of relative CAN addresses. In such a case, there may be a base address that must be applied to the RCAs of all monitor and control points. In this case, the base address is entered as a Hex-formatted number. If there is no such base address, “none” is entered. If the base address must be supplied as a parameter, “parm” is entered.

3.1.11. ICD

Valid values

- **Text**

Discussion

This cell should contain the name of the ICD on which this spreadsheet is based. For example, ALMA-42.02.00.00-75.35.25.00-A-ICD is the name of the Holography ICD.

3.1.12. ICD Date

Valid values

- **Text** (*A valid date*)

Discussion

This cell should contain the date associated with the ICD on which this spreadsheet is based, for example, 2004-07-27. In Excel, if a valid date is entered as “2004-07-27”, it will be reformatted for the display as 7/27/2004. However, internally it is stored as “2004-07-27T00:00:00.000”.

3.2. Monitor

The column names in the Monitor spreadsheet must be:

- Assembly
- Name

- RCA (CAN only)
- Raw Data Type (CAN only)
- TE Related (CAN only)
- World Data Type (CAN only)
- Data Type (Ethernet only)
- Data Units
- Scale (CAN only)
- Offset (CAN only)
- Min Range
- Max Range
- Default
- Error Condition
- Error Severity
- Error Action
- Mode
- Implement
- External
- Can Be Invalid (CAN only)
- Description

The first row of the Monitor spreadsheet must be the name of this spreadsheet: "Monitor Point." The second row contains the names of the columns. Beginning with the third row each monitor point is entered: one for each row. This and all subsequent rows must contain a fixed number of columns.

3.2.1. Assembly

Valid values

- **Name**

Discussion

The name in this cell must refer to a name in column 1 of the Main spreadsheet, i.e. it must be the name of the assembly to which this monitor point belongs.

3.2.2. Name

Valid values

- **^Name**
- **Name**

Discussion

This cell should contain the name of the monitor point. The notation '^' preceding a name denotes a "dependent" name, one that depends on a previously defined monitor point. The '^' character points to the previous rows; the name on which this monitor point depends is the preceding row that does not contain a beginning '^' character. In other words a monitor point is immediately followed by its dependents, if it has any. This notation is used to expand status bytes into, for example, one or more bits that denote Boolean properties. This topic is discussed in more detail in section 5.

The name of the monitor point must be unique within the Monitor spreadsheet, even if the monitor points belong to different assemblies within the LRU. Usually the name of the monitor point should be exactly as it appears in the ICD. However, the "GET_" prefix is dropped in forming the name of the monitor point. Naming is discussed in section 6.

3.2.3. RCA (CAN only)

Valid values

- **none**
- **Hex**

Discussion

This cell should contain the value of the relative CAN address in hex notation, as specified in the ICD. A dependent monitor point should have the same RCA as the monitor point on which it depends. The value 'none' is also permitted here for those cases that are intended to declare methods that are to be implemented in the extended class.

3.2.4. Raw Data Type (CAN only)

Valid values

- **bit**
- **int8**
- **uint8**
- **int16**
- **uint16**
- **int24**
- **uint24**
- **int32**
- **uint32**

- **int48**
- **uint48**
- **int64**
- **uint64**
- **float**
- **double**
- **string**
- ***dependent-monitor-point-notation*** (explained in section 5)

Discussion

The raw data type is the type associated with the data as it is represented internally in the hardware device. For example, a voltage may be represented internally as a signed 16-bit integer with some associated scaling factor. The following list discusses each valid raw data type.

- **bit** A single bit within a ubyte. Unused bits within ubytes are padded to fill the byte.
- **int8** A signed 8 bit integer value in two's complement format.
- **uint8** An unsigned 8 bit integer value.
- **int16** A signed 16 bit integer value in two's complement format. The most significant byte is transmitted first.
- **uint16** An unsigned 16 bit integer value. The most significant byte is transmitted first.
- **int24** A signed 24 bit integer value in two's complement format. The most significant byte is transmitted first.
- **uint24** An unsigned 24 bit integer value. The most significant byte is transmitted first.
- **int32** A signed 32 bit integer value in two's complement format. The most significant byte is transmitted first.
- **uint32** An unsigned 32 bit integer value. The most significant byte is transmitted first.
- **int48** A signed 48 bit integer value in two's complement format. The most significant byte is transmitted first.
- **uint48** An unsigned 48 bit integer value. The most significant byte is transmitted first.
- **int64** A signed 64 bit integer value in two's complement format. The most significant byte is transmitted first.
- **uint64** An unsigned 64 bit integer value. The most significant byte is transmitted first.
- **float** Single precision 32 bit IEEE floating point value. The most significant byte, containing the sign bit, is transmitted first.
- **double** Double precision 64 bit IEEE floating point value. The most significant byte, containing the sign bit, is transmitted first.
- **string** A string of single byte characters. Length is given by the DLC field in the CAN frame and the string is not null terminated. The differences between a string and a ubyte[n] is that for strings n is not specified, and each byte is to be treated as an ASCII character.

Array notation may be used on any item except string. For example, ubyte[n] is a stream of ubytes, exactly n ubytes long. The first byte transmitted on the bus is ubyte[0] and the last byte transmitted on the bus is ubyte[n-1].

3.2.5. TE Related (CAN only)

Valid values

- **yes**
- **no**

Discussion

This cell identifies whether this monitor point is related to a timing event or not. All ICDs are required to state this information.

3.2.6. World Data Type (CAN only)

Valid values

• **ubyte**	*unsigned 8-bit byte*
• **short**	*signed 16-bit integer*
• **ushort**	*unsigned 16-bit integer*
• **int**	*signed 32-bit integer*
• **uint**	*unsigned 32-bit integer*
• **long**	*signed 64-bit integer*
• **ulong**	*unsigned 64-bit integer*
• **float**	*IEEE float*
• **double**	*IEEE double*
• **boolean**	*0 or 1*
• **string**	*sequence of ACSII characters*
• **acstime**	*ACS::Time*
• **void**	*no specified type*
•	

Discussion

The world data type is the type associated with the data as it is represented in the external world. For example, the raw data type for a voltage may be a scaled, signed 16-bit integer, but its world data type is a float, representing the data in volts.

Array notation may be used on any item except string. For example, ubyte[n] is a stream of ubytes, exactly n ubytes long. The first byte transmitted on the bus is ubyte[0] and the last byte transmitted on the bus is ubyte[n-1].

3.2.7. Data Units

Valid values

- **none**
- **decibel** *only allowed as a dimensionless unit*
- **kilogram**
- **second**
- **ampere**
- **kelvin**
- **radian**
- **steradian**
- **hertz**
- **watt**
- **coulomb**
- **volt**
- **ohm**

Discussion

Data units are SI and all data values stored in the monitor database are in SI units except timestamps, which are internal to the database. This list may be extended in the future. The value “none” is used to denote that no units apply to this quantity.

3.2.8. Scale (CAN only)

Valid values

- **none**
- **extended**
- **CelsiusToKelvin**
- **Number**

Discussion

Scale and offset provide a means for converting between raw and world values. The conversion formula is:

world-value (in the specified units) = raw-value * scale + offset

This represents the most common case in the ICDs. If no conversion is required “none” is entered. There are some more complex cases where conversion is required that are not covered by this simple formula. For these cases, “extended” is entered and the conversion routine must be implemented in the extended class. A lot of raw temperature units are in Celsius. The SI unit of temperature is the Kelvin. The “CelsiusToKelvin” entry handles this conversion. (Remember that the formula is “Kelvin = CelsiusToKelvin(raw)”.)

3.2.9. Offset (CAN only)

Valid values

- **none**
- **extended**
- **Number**

Discussion

See the discussion for ‘Scale’.

3.2.10. Min Range

Valid values

- **none**
- **Number**
- **Hex**

Discussion

The minimum and maximum ranges are entered either as a decimal number or as a hexadecimal number. The latter is convenient for specifying bit patterns. If no range is specified, “none” is entered.

3.2.11. Max Range

Valid values

- **none**
- **Number**
- **Hex**

Discussion

See the discussion for ‘Min Range’.

3.2.12. Default

Valid values

- **none**
- **mid**
- **random**
- **Number**
- **Hex**

Discussion

This cell contains the value of this monitor point that is to be used within simulations. If nothing is to be entered “none” is entered. Default values for sequences of any kind are not allowed. The value “mid” is the value at the midpoint between the minimum and maximum values. The value “random” is a random value that lies between the minimum and maximum values. A fixed number may also be supplied.

3.2.13. Error Condition

Valid values

- **none**
- **Text**

Discussion

If there is no error condition associated with this monitor point, “none” should be entered. Otherwise, the text that should be entered is a logical condition, usually related to the raw data associated with the monitor point. For example, suppose the monitor point is a status bit and a value of ‘1’ indicates a severe error. Then, the raw data type is bit, the world data type is ‘boolean’, and the error condition is ‘true’. This is probably the simplest case; but, in general, the text can represent more complex logical conditions.

3.2.14. Error Severity

Valid values

- **none**
- **error**
- **severe**

Discussion

At present, “error” and “severe” represent the levels of errors. If there is no error condition associated with the monitor point, “none” should be entered.

3.2.15. Error Action

Valid values

- **none**
- **continue**
- **stop**
- **extended**

Discussion

If an error condition is entered, this column denotes the action that is to be taken in the event the error is triggered. First, the error sub-state of the device is turned on. Second, the error is always reported to the parent of the device. Then the action stated here is taken. The value ‘continue’ means to continue operating the device, although in an error state. The value ‘stop’ means to place the device in a stopped state, which means that the device will no longer function until it is re-started (which means it must be re-configured and re-initialized). The value ‘extended’ means that the action to be taken is specified in a method that is implemented in the extended class.

3.2.16. Mode

Valid values

- **operational**
- **diagnostic**
- **startup**

Discussion

These values are discussed in the introduction.

3.2.17. Implement

Valid values

- **yes**
- **no**

Discussion

This value designates whether this method should be implemented by the code generation framework or not. If it is not, the method is only declared and it must be implemented in the extended class.

3.2.18. External

Valid values

- **yes**
- **no**

Discussion

The ‘External’ column designates whether this monitor point should be accessible via a CORBA client or not. If it is, then an IDL definition is created to support such access.

3.2.19. Can Be Invalid (CAN only)

Valid values

- **yes**
- **no**

Discussion

This column is only ‘yes’ for the FOAD and may disappear in the future. It means that a value from the hardware may be invalid (which is detected by its being a negative number), in which case it must be immediately read again.

3.2.20. Description

Valid values

- **Text**

Discussion

This cell should contain a description of this monitor point.

3.3. Control

The column names in the Control spreadsheet must be:

- Assembly
- Name
- RCA (CAN only)
- Address (Ethernet only)
- Raw Data Type (CAN only)
- TE Related (CAN only)
- World Data Type (CAN only)
- Data Type (Ethernet only)
- Data Units
- Scale (CAN only)
- Offset (CAN only)
- Min Range
- Max Range
- Returns (CAN only)
- Parameter (CAN only)
- Data (CAN only)
- Value (CAN only)
- Mode (CAN only)
- Implement
- External
- Description

3.3.1. Assembly

This cell is the same as Assembly in the Monitor section.

3.3.2. Name

- **Name**

This cell contains the name of the control point. The name must be unique within the Control spreadsheet, even if the control points belong to different assemblies within the LRU. Usually the name of the control point should be exactly as it appears in the ICD. However, the “SET_” prefix is dropped in forming the name of the control point. Naming is discussed in section 6.
Note: Control points cannot have dependent control points.

3.3.3. RCA (CAN only)

This cell is the same as RCA in the Monitor section.

3.3.4. Raw Data Type (CAN only)

This cell is the same as Raw Data type in the Monitor section.

3.3.5. TE Related (CAN only)

This cell is the same as TE Related in the Monitor section.

3.3.6. World Data Type (CAN only)

This cell is the same as World Data Type in the Monitor section.

3.3.7. Data Units

This cell is the same as Data Units in the Monitor section.

3.3.8. Scale (CAN only)

This cell is the same as Scale in the Monitor section.

3.3.9. Offset (CAN only)

This cell is the same as Offset in the Monitor section.

3.3.10. Min Range

This cell is the same as Min Range in the Monitor section.

3.3.11. Max Range

This cell is the same as Max Range in the Monitor section.

3.3.12. Returns (CAN only)

Valid values

- **void**
- **Text**

Discussion

The signature of what is returned by the method. This is usually ‘void’.

3.3.13. Parameter (CAN only)

Valid values

- **void**
- **default**
- **Text**

Discussion

This indicates the type associated with the argument of the method. There is a special value that can be inserted: ‘default’ means that the type is indicated by the world data type (which is usually the case). The reason this column exists is for an escape mechanism. For a complex method, this column may specify more than one parameter. Anything (other than ‘default’) is merely copied to the method parameter list.

If ‘default’ is specified and the control point is TE related, two arguments are generated: the world data type and a requested time.

3.3.14. Data (CAN only)

Valid values

- **parm**
- **fixed**

Discussion

This column is either ‘parm’ or ‘fixed’. If the cell contains ‘fixed’, this means that the method uses a fixed value that is specified in the Value column and that its parameter is void. If the column is ‘parm’ this means that the signature of the method is specified by the ‘Parameter’ column.

3.3.15. Value (CAN only)

Valid values

- **0**
- **Number**
- **Hex**

Discussion

This is the value that must be used if the ‘Data’ column is fixed. If ‘Data’ is not fixed this column is ignored.

3.3.16. Mode

This cell is the same as Mode in the Monitor section.

3.3.17. Implement

This cell is the same as Implement in the Monitor section.

3.3.18. External

This cell is the same as External in the Monitor section.

3.3.19. Description

This cell is the same as Description in the Monitor section.

3.4. Archive

The column names in the Archive spreadsheet must be:

- Assembly
- Name
- Refers To
- Interval
- Only On Change
- Display Units
- Graph Min
- Graph Max
- Format

- Title

3.4.1. Assembly

This cell is the same as Assembly in the Monitor section.

3.4.2. Name

- **Name**

This cell contains the name of the archived property as it appears in the monitor database.

3.4.3. Refers To

Valid values

- **none**
- **Name**
- **^Name**

Discussion

The ‘Refers To’ column can only refer to a monitor point. So, the name must indicate a monitor point name in column two of the monitor spreadsheet. The value ‘none’ is only used if the archive property refers to logging the value of a control point.

3.4.4. Interval

Valid values

- **none**
- **Number**
- **^Name**
- **te/Number**

Discussion

• **none**	*The value ‘none’ is used for dependent bits or array elements.*
• **Number**	*The value “Number” is a sampling interval in seconds.*
• **^Name**	*The value “^Name” must refer to a control point and means that a value is inserted in the monitor database when this control point is called.*
• **te/Number**	*The value “te/Number” means that this property is sampled at every timing event but is only written to the archive every “Number” seconds.*

3.4.5. Only On Change

Valid values

- **yes**
- **no**

Discussion

This column designates whether this property should be inserted into the monitor database every time it is sampled or only when its value changes.

3.4.6. Display Units

Valid values

- **none**
- **Celsius**
- **decibel**
- **SI prefixes for multiples and submultiples applied to World Data Units.**

Discussion

This cell must be consistent with the associated World Data Type and may contain standard SI prefixes for multiples and submultiples that are applied to that basic type. Supported SI prefixes are:

- deci
- centi
- milli
- micro
- nano
- pico
- femto
- atto
- deca
- hecto
- kilo
- mega
- giga
- tera
- peta
- exa

3.4.7. Graph Min

Valid values

- **none**
- **Number**

Discussion

The ‘Graph Min’ and ‘Graph Max’ columns should contain the scale on the “value axis” of a plot of value vs. time. The ‘Format’ value is the C “printf” quantity needed to convert a binary value to a string, and the ‘Title’ value is the title to be displayed on the plot.

3.4.8. Graph Max

Valid values

- **none**
- **Number**

Discussion

See ‘Graph Min’.

3.4.9. Format

Valid values

- **Text**

Discussion

See ‘Graph Min’.

3.4.10. Title

Valid values

- **Text**

Discussion

See ‘Graph Min’.

4. Differences between CAN-bus and Ethernet devices

A recent addition to the code generator frame work were the already above mentioned Ethernet devices. It has been tried to keep the abstraction level for CAN-bus and Ethernet-devices the same, but some changes were necessary nonetheless. It has been mentioned above that the Main page contains the name of a "Vendor Class" which specifies the file names for a C++-class which implements the communication layer with the Ethernet-device. The implementation is independent of all code in the CONTROL subsystem but for one module: CONTROL/Common/EthernetDevice. It is the communication layer's responsibility to implement the interface EthernetDeviceInterface by inheritance and all of its pure virtual methods:

- A name service lookup of the host name. The name has to be provided in the CDB;
- Opening of a TCP or UDP socket to the IP address of the device. The port has to be provided in the CDB.
- Set up of the socket parameters. The CDB contains "lingerTime" and "timeout" attributes that have to be set for the socket.
- Optional: comparison of the device MAC address with the one provided in the CDB in order to avoid any misconfigurations.
- Data transfer from and to the device. The CDB contains an attribute "replies" giving the number of additional retries after the first failed attempt.
- Proper error handling via exceptions. Allowed exceptions are described in the file EthernetDeviceInterface.h.
- Optional: Addresses are in the spreadsheet usually given as abstract positive (including 0) integer numbers. The VendorClass can define an enumeration which has public visibility and contains a list of name-address pairs of which the names can be entered in the Address field instead

An example for a name of a Vendor Class on the main page could be "MyEth". This translates to a class of the name MyEth that implements the device communication in the two files include/MyEth.h and src/MyEth.cpp which are both not code generated and will never be overwritten.

The Monitor page does not contain any scaling information because Ethernet-devices send all values already "world-ready", e.g. if a device reads internally something that becomes in the real world 10.7e9Hz, it will return a value of 10.7e9. The same holds true for the Control page where again no scaling factors or offsets are to be given. All values are expected to be real-world values which will be converted by the hardware internally. Also the specification of the input parameters is obsolete since only a maximum of one parameter per Control Point is allowed.

One important difference between CAN-bus and Ethernet-devices to keep in mind is that CAN-bus devices support TE-related monitor and control requests but Ethernet-devices do not and cannot.

5. Names in Generated Code

Suppose we have a monitor point in the ICD whose name is GET_SUPPLY_CURRENT and a control point in the ICD whose name is SET_SUPPLY_CURRENT. (All names in the ICD are supposed to begin with 'GET_' and 'SET_'.)

We adopt the following naming conventions in the generated code. These conventions make the CORBA and CCL method names exactly like the monitor and control point names listed in the ICD.

The names recorded in the Monitor and Control spreadsheets drop the 'GET_' and 'SET_' prefixes on the monitor and control point names.

5.1. Monitor Point

5.1.1. Archive Name

The name of this monitor point in the archive is **`SUPPLY_CURRENT`**.

5.1.2. IDL

There are two IDL quantities defined: the BACI property **`SUPPLY_CURRENT`** and the method seen by the CCL **`GET_SUPPLY_CURRENT`**. Their definitions are as follows:

```
readonly attribute ACS::ROfloat SUPPLY_CURRENT;
float GET_SUPPLY_CURRENT(out ACS::Time timestamp)
    raises(ControlExceptions::CAMBErrorEx, ControlExceptions::INACTErrorEx);
```

5.1.3. C++ .h

There are three methods defined in C++, two are public and one is protected. The public methods are **`SUPPLY_CURRENT`**`()`, that implements the BACI property, and **`GET_SUPPLY_CURRENT`**`(Time & timestamp)`, that implements the IDL method. The protected method is internal to the C++ implementation and is **`getSupplyCurrent`**`(Time & timestamp)`. Their definitions are as follows:

```
virtual ACS::ROfloat_ptr SUPPLY_CURRENT();
virtual CORBA::Float GET_SUPPLY_CURRENT(Time & timestamp)
;
virtual float getSupplyCurrent(Time & timestamp)
;
```

5.2. Control Point

5.2.1. Archive Name

If there is a requirement to archive the new value whenever this control point is changed, its name in the archive is **SET_SUPPLY_CURRENT**.

5.2.2. IDL

The IDL method that allows one to access the control point from the CCL is **SET_SUPPLY_CURRENT**. Its definition is as follows:

```
void SET_SUPPLY_CURRENT(in float world)
    raises(ControlExceptions::CAMBErrorEx, ControlExceptions::INACTErrorEx);
```

5.2.3. C++ .h

There are two C++ methods defined, a public one that implements the IDL method **SET_SUPPLY_CURRENT**, and a protected one that is internal to the C++ implementation, **setSupplyCurrent**. Their definitions are as follows:

```
virtual void SET_SUPPLY_CURRENT(CORBA::Float world)
;
virtual void setSupplyCurrent(float world)
;
```

6. Structure of the Code Generator

This section outlines the structure of the code generation templates for the generated C++ header files and IDL files. Its purpose is to aid in reading the template and understanding its logical structure.

The following are logical conditions governing the generated code.

- isExternal
  Is the monitor or control point to be accessed by external CORBA clients?
- isMonitored
  Is the monitor point periodically monitored (by a BACI thread)?
- isConversion
  Does the monitor or control point require conversion between raw and world data?
- isSpecialConversion
  *Is the conversion required linear, i.e. of the simple "world = raw * scale + offset" type?*
- isLinked
  Do the monitor and control point share the same RCA and attributes?
- isTERelated
  Is the monitor or control point related to a timing event?
- isDependent
  Is the monitor a dependent value?
- hasDependents
  Does this monitor point have dependents?
- isImplemented
  Is the method for this monitor or control point to be automatically implemented in the generated base class?

6.1. Monitor Point

ICD Name: **GET_SUPPLY_VOLTAGE**
Spreadsheet Name: **SUPPLY_VOLTAGE**
Archive Name: **SUPPLY_VOLTAGE**

6.1.1. Generated IDL:

If isExternal and isMonitored

```
        readonly attribute ACS::ROfloat SUPPLY_VOLTAGE;
        float GET_SUPPLY_VOLTAGE(out ACS::Time timestamp)
                raises(ControlExceptions::CAMBErrorEx,
                 ControlExceptions::INACTErrorEx);
      long GET_RCA_SUPPLY_VOLTAGE();
```
Endif

6.1.2. Generated C++ .h:

<u>Private section</u>

a. Field for RCA
```
      long rcaSupplyVoltage;
```
b. Fields for conversion
If isConversion and !isSpecialConversion
```
      double scaleSupplyVoltage;
      double offsetSupplyVoltage;
```
Endif
c. Fields needed for dependent monitor points
If isDependent
```
      float valueSupplyVoltage;
      ACS::Time timeSupplyVoltage;
```
Endif
d. Any other fields
If isExternal and isMonitored
```
      SmartPropertyPointer< ROfloat > sppSupplyVoltage;
```
Endif

<u>**Protected section**</u>

a. Major monitor method
If isImplemented
      If isDependent
```
      virtual float getSupplyVoltage(ACS::Time& timestamp)
       {
            timestamp = timeSupplyVoltage;
            return valueSupplyVoltage;
      }
```
      Else
```
    virtual float getSupplyVoltage(ACS::Time& timestamp)
;
```
      Endif
Else
```
      virtual float getSupplyVoltage(Time & timestamp)
       = 0;
```
Endif
b. Get and set methods for RCA
```
      virtual int getRCASupplyVoltage() const
      {
            return rcaSupplyVoltage;
      }
      virtual void setRCASupplyVoltage(int rca)
      {
```

```
                rcaSupplyVoltage = rca;
        }
```
c. Conversion methods

If isConversion

```
        If !isSpecialConversion
        virtual float rawToWorldSupplyVoltage(unsigned short raw) const;
        virtual double getScaleSupplyVoltage() const
        {
                return scaleSupplyVoltage;
        }
        virtual double getOffsetSupplyVoltage() const
        {
                return offsetSupplyVoltage;
        }
        virtual void setConversionSupplyVoltage(double scale, double offset)
        {
                scaleSupplyVoltage_scale = scale;
                offsetSupplyVoltage = offset;
        }
        Else
        virtual float rawToWorldSupplyVoltage(unsigned short raw) const = 0;
        Endif
```
Endif

Public section

If isExternal and isMonitored

a. Major monitor method

```
        virtual CORBA::Float GET_SUPPLY_VOLTAGE(ACS::Time& timestamp)
        {
                return getSupplyVoltage(timestamp);
        }
```
b. Get method for RCA

```
        virtual CORBA::Long GET_RCA_SUPPLY_VOLTAGE
        {
                return getRCASupplyVoltage();
        }
```
c. Method for monitored property

```
        virtual ACS::ROfloat_ptr SUPPLY_VOLTAGE();
```
Endif

6.2. Control Point

ICD Name: **SET_SUPPLY_VOLTAGE**

Spreadsheet Name: **SET_SUPPLY_VOLTAGE**

Archive Name: **SET_SUPPLY_VOLTAGE**

6.2.1. Generated IDL:

```
If isExternal
        If isTERelated
        void SET_SUPPLY_VOLTAGE(in float world, in Time requestTime)
                raises(ControlExceptions::CAMBErrorEx,
                        ControlExceptions::INACTErrorEx);
        Else
        void SET_SUPPLY_VOLTAGE(in float world)
                raises(ControlExceptions::CAMBErrorEx,
                        ControlExceptions::INACTErrorEx);
        Endif

        long GET_RCA_SET_SUPPLY_VOLTAGE();
Endif
```

6.2.2. Generated C++ .h:

Private section

a. Field for RCA

```
        long rcaSetSupplyVoltage;
```

b. Fields for conversion

```
        If isConversion and !isSpecialConversion
        double scaleSetSupplyVoltage;
        double offsetSetSupplyVoltage;
        Endif
```

c. Fields needed for archiving

```
If isArchived and isOnlyOnChange
        float previousValueSetSupplyVoltage;
Endif
```

Protected section

a. Major control method

```
If isImplemented

        If isTERelated
        virtual void setSupplyVoltage(float world, ACS::Time& requestTime)
;
        Else
        virtual void setSupplyVoltage(float world)
;

        Endif
Else
        If isTERelated
        virtual void setSupplyVoltage(float world, ACS::Time& requestTime)
 = 0;
```

```
        Else
        virtual void setSupplyVoltage(float world)
 = 0;
        Endif
Endif
b. Get and set methods for RCA
        If !isLinked
        virtual int getRCASetSupplyVoltage() const
        {
              return rcaSetSupplyVoltage;
        }
        virtual void setRCASetSupplyVoltage(int rca)
        {
              rcaSetSupplyVoltage = rca;
        }
        Endif
c. Conversion methods
If isConversion
        If !isSpecialConversion
        virtual unsigned short worldToRawSupplyVoltage(float world) const;
        virtual double getScaleSetSupplyVoltage() const
        {
              return scaleSetSupplyVoltage;
        }
        virtual double getOffsetSetSupplyVoltage() const
        {
              return offsetSetSupplyVoltage;
        }
        virtual void setConversionSetSupplyVoltage(double scale,
              double offset)
        {
              scaleSetSupplyVoltage_scale = scale;
              offsetSetSupplyVoltage = offset;
        }
        Else
        virtual unsigned short worldToRawSupplyVoltage(float world) const = 0;
        Endif
Endif

public
If isExternal
a. Major control method

        If isTERelated
        virtual void SET_SUPPLY_VOLTAGE(CORBA::Float world,
              ACS::Time& requestTime)
         {
              setSupplyVoltage(world, requestTime);
        }
        Else
        virtual void SET_SUPPLY_VOLTAGE(CORBA::Float world)
         {
```

```
        setSupplyVoltage(world);
    }
```

Endif

b. Get method for RCA

```
    virtual CORBA::Long GET_RCA_SET_SUPPLY_VOLTAGE
    {
        return getRCASetSupplyVoltage();
    }
```

Endif

Notes:

- *These conventions make the CORBA and CCL method names be exactly like the monitor and control point names in the ICD.*
- *Names in the spreadsheet drop the "GET_" prefixes on monitor points.*
- *If you are reading the code generation templates, the only additional attributes needed are related to either the raw or world data type being an array.*

7. Status Bits and Dependent Monitor Points

A monitor point is independent if it is sampled by executing a single command. The raw data type of an independent monitor point may be a single value or an array of values. A dependent monitor point is one that is derived from an independent monitor point.

There are two cases that are relevant to this discussion. One is the case of a status bits within one or more bytes. The other is an array of values that are read by a single command. Values are not stored in the monitor database as arrays; the individual elements of the array are named and stored in the database.

The notation describing dependent monitor points (those beginning with '^Name') is included in the monitor spreadsheet in the column labeled "Raw Data Type". The following notation is supported, where 'i', 'j', 'k', 'n', and 'm' are integer numbers.

• <i>	*denotes the i^{th} bit within a byte*
• <i><j>	*denotes the j^{th} bit within the i^{th} byte*
• <i – j>	*denotes bits i through j within a byte*
• <i><j – k>	*denotes bits j through k within the i^{th} byte*
• [n]	*denotes the n^{th} element of an array*
• [n – m]	*denotes element n through m of an array*

If the notation in the "Raw Data Type" column indicates one or more bits, then the "World Data Type" is limited to being a "boolean" or "int".