AOCS UNIT MANAGEMENT FRAMELET

Concept And Architecture Description

Abstract

This document was written as part of the study "Design and Prototyping of a Software Framework for the AOCS" done under contract Estec/13776/99/NL/MV for ESA-Estec. The purpose of the study is the development of a software framework for the Attitude and Orbit Control Subsystem (AOCS) of a satellite. The framework will be built as a collection of framelets. This document describes the AOCS unit management framelet. This framelet defines an architecture to manage external AOCS units. The framelet enhances reusability because it provides a standard interface for AOCS units that decouples the managers and users of unit data, from the units themselves.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.3

Reference: SWE/99/AOCS/017

TABLE OF CONTENTS

1 REFERENCES

RD1 E. Gamma *et al.* (1995), *Design Patterns – Elements of Reusable Object Oriented Software,* Reading, Massachusetts: Addison-Wesley

RD2 A. Pasetti (2000), *AOCS Framework – Concept Level Description*, AOCS Framework Document ref. SWE/99/AOCS/004

RD3 *Modular Attitude Control System (MACS) Handbook*

RD4 A. Pasetti (2000), *AOCS Framework - Prototype Definition*, AOCS Framework Document ref. SWE/99/AOCS/019

RD5 A. Pasetti (2000), *AOCS Framework – AOCS Prototype Definition*, AOCS Framework Document ref. SWE/99/AOCS/020

RD6 A. Pasetti (2000), *AOCS Framework – Methodological Issues*, AOCS Framework Document ref. SWE/99/AOCS/018

2 ACRONYMS

AAD	Attitude Anomaly Detection
AOCS	Attitude and Orbit Control Subsystem
AST	Autonomous Star Tracker
CSS	Coarse Sun Sensor
ES	Earth Sensor
FDIR	Failure Detection, Isolation and Recovery
FPM	Fine Pointing Mode
FSS	Fine Sun Sensor
GYR	Gyroscope
KF	Kalman Filter
IAM	Initial Acquisition Mode
OBDH	On-Board Data Handling system (aka as OBDS)
NM	Normal Mode
NTT	Non-Time-Tagged
OCM	Orbit Control Mode
OO	Object-Oriented
PD	Proportional-Derivative controller
PI	Proportional-Integral controller
PID	Proportional-Integral-Derivative controller
RRM	Rate Reaction Mode
RTOS	Real-Time Operating System
RW	Reaction Wheel
SAS	Sun Attitude Sensor
SBM	Stand-By Mode
SPS	Sun Presence Sensor
STR	Star Tracker
SLM	Slewing Mode
SM	Safe Mode
TC	Telecommand
THU	Thruster
TM	Telemetry
TT	Time-Tagged

3 INTRODUCTION

This document describes the *AOCS unit management framelet* for the AOCS framework. The framelet is described at both the framelet concept level and at the framelet architectural level.

This framelet defines an architecture to manage external AOCS units. The framelet enhances reusability because it provides a standard interface for AOCS units that decouples the managers and users of unit data, from the units themselves.

3.1 Context

The context for the design of the framelet is described in RD2. The present document assumes that the reader is familiar with RD2 and in particular with the sections dealing with AOCS unit management and with the fictitious unit management.

The architecture proposed here follows the concept outlined in RD2.

In comparing the present document with RD2, it should be kept in mind that the class definitions presented in the latter document are not necessarily entirely consistent with the class definitions presented here. This is because the main purpose of RD2 was to introduce an architectural *concept* whereas the main purpose of the present document is to describe an architecture. The design presented here therefore should be regarded as an evolution of the design presented in RD2.

3.2 Applicability to Java Version

The AOCS Framework was first implemented in C++ and then ported to Java. This document was originally written for the C++ version. Generally speaking, the description of the framelet at design level – in particular its design patterns – is language-independent and is equally applicable to both the C++ and Java versions whereas the architectural-level description is more tied to the C++ version. For a detailed description of the architecture of the Java framework, readers should refer to the JavaDoc documentation generated from it.

The porting of the AOCS Framework to Java was done in the "Real Time Java Project". The issues that should be borne in mind when using this document for the Java version of the AOCS framework are presented in the project web site currently located at the following address: www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html..

3.3 Notation

The pseudo-code examples in this document use a C++ notation.

UML class diagrams were obtained with the *Together* tool.

4 FRAMELET CONSTRUCTS

The architectural constructs exported by this framelet are listed in the following table:

AOCS UNIT FRAMELET
Design Patterns
Fictitious Unit Pattern : pattern to make objects that process unit data look like units
Framelet Interfaces and Abstract Base Classes
`AocsUnitHardware` : interface for objects managing low level exchanges with external units. `UnitInstruction` : interface structure defining a generic protocol for data exchanges with external units `AocsUnitFunctional` : interface for objects representing the functional exchanges between the AOCS software and an AOCS unit (either real or fictitious) `AocsUnitHousekeeping` : interface for objects representing the housekeeping exchanges between the AOCS software and a real (ie. non fictitious) AOCS unit `AocsUnit` : abstract class serving as base class for all objects representing external unit proxies in the AOCS software `TriggerList` : interface for trigger list objects, namely list of units due to be triggered at the same time in the AOCS cycle
Framelet Core Components
`PollingTrigger` : trigger object to perform full data transfer (transaction + refresh cycle) with polling on registered units `UnitTrigger` : trigger object to perform full data transfer (transaction + refresh cycle) without polling on registered units `RefreshTrigger` : trigger object to perform refresh operations on registered units `TrasactionTrigger` : trigger object to perform transaction operations on registered units
Framelet Default Components
`FullTriggerList` : full implementation of interface `TriggerList` `FunctionalTriggerList` : partial implementation of interface `TriggerList` covering only

functional units

`MacsTcController` : implementation of interface `AocsUnitHardware` for a MACS telecom controller

`FssPrototype` : two-axis fine sun sensor AOCS unit for the AOCS prototype

`GyrPrototype` : single-axis gyro AOCS unit for the AOCS prototype

`RwPrototype` : reaction wheel AOCS unit for the AOCS prototype

`SapPrototype` : solar acquisition and propulsion electronics AOCS unit for the AOCS prototype

`TorquingThrusters` : fictitious AOCS unit to command a set of thrusters directly with the torque requests around spacecraft axes

The components listed above are those envisaged for the prototype version of the AOCS framework. Later versions may offer a richer set of default implementations of the framelet interfaces. In particular, more hardware interface and AOCS unit objects might be offered.

5 AOCS UNIT MODEL

The AOCS framework assumes that units are *passive*, namely incapable of initiating data exchanges with the AOCS computer. All data transactions with a unit occur in response to a command from the AOCS computer.

Data transactions between the AOCS computer and the AOCS units fall under two categories:

- *Functional Transactions*

  This type of transaction is cyclical and relates to the primary function for which a unit was designed. Thus, for instance, a thruster unit periodically receives firing profiles, a gyro periodically supplies rate information, a reaction wheel periodically receives torque requests.

- *Housekeeping Transactions*

  This type of transaction may be occasional and it consists of commands sent by the AOCS computer to the unit (eg. to change its operational mode, to switch the unit on or off, etc) and of housekeeping data sent by the unit to the AOCS computer (eg. temperature readings, results of self-tests)

The unit status is defined by:

- *Power Status*

  Units can either be “power on” or “powered off”

- *Operational Mode*

  Units may be in one of several operational modes

- *Health Status*

  Units can either be “healthy” or “unhealthy”

Finally, units may be able to perform a *self-test*. The result of a self-test is reported as an integer.

5.1 Unit Data Formats

Data in the AOCS units can exist at two levels of abstraction:

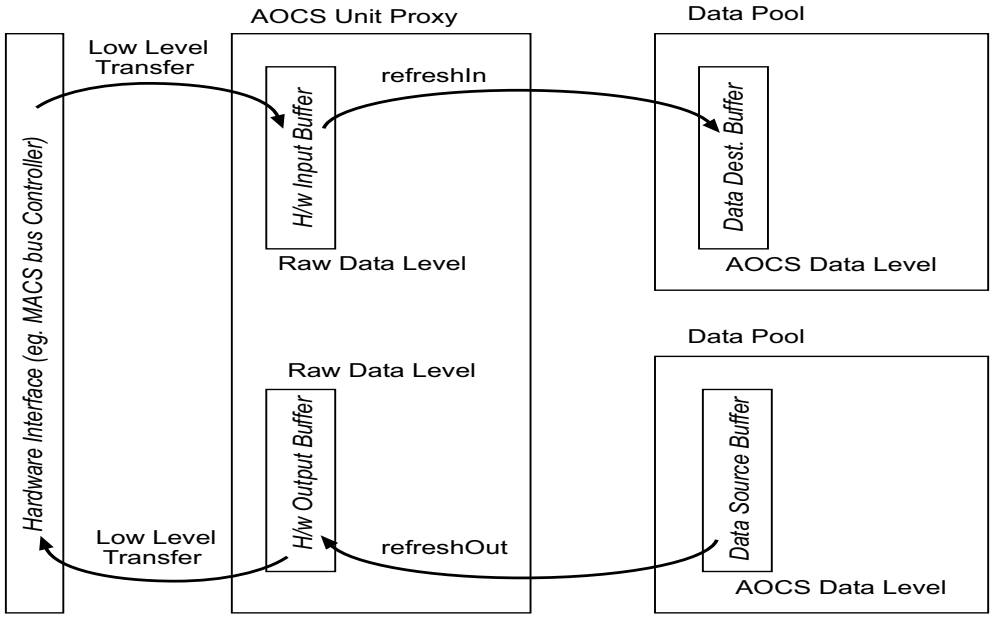
- the *raw data level* representing the data as they are transmitted on the physical communication link between the unit and the AOCS computer;

- the *AOCS data level* representing the data as they are used by clients of the AOCS unit objects in the AOCS software. Such data are expressed in engineering units and, if they depend on reference frame, are expressed in the spacecraft reference frame.

The raw data level is mission-specific and cannot be defined here. This data level, however, is internal to the AOCS unit components. All exchanges between the AOCS unit components and other AOCS software objects take place at the AOCS data level. It is the responsibility of the AOCS unit object to perform conversions between raw and AOCS data levels.

5.2 Data Exchange Model

The model for the exchange of data between hardware and AOCS unit objects is illustrated in the figure below. The figure assumes the case of a unit that can both receive and send data (this could for instance be the case of a reaction wheel that receives torque requests and sends wheel velocity measurements). Many units will only work in receive or send mode.



As shown in the figure, the AOCS unit object maintains *hardware buffers* where incoming or outgoing data are deposited at raw data format. The transfer between these buffers and the hardware interface is done by low level mechanisms.

In a typical example. The hardware interface could be a MACS bus controller. Data reception triggers an interrupt whose servicing routine deposits the incoming data word in the hardware input buffer where it remains available for further processing by the AOCS unit. Data to be sent to the unit must instead be written to an I/O address. In that case, the hardware output buffer serves as the source for the data written to the I/O port by a dedicated routine.

Only single hardware buffers are shown in the figure. In reality, the buffers may be made up of several memory locations holding related data. The hardware output buffer for a thruster, for instance, will consist of at least two locations holding the firing duration and firing delay data.

The AOCS unit object maintains links to data pool locations where the incoming and outgoing data are stored as variables at AOCS data level. The location where incoming data are stored is called *destination data buffer* and the location from which outgoing data are retrieved is called *source data buffer*. The destination and source data buffers are sets of data items in the data pools.

In the case of a sun sensor, for instance, the destination data buffer is made up of two (or, depending on the representation, three) data items representing the sun vector as measured by the sun sensor but expressed in engineering units and referred to the spacecraft coordinate frame.

When the data to be sent by a unit represent a measurement performed by the unit, it may be necessary to send a *synchronization command* to the unit. This command directs the unit to acquire the measurement and store it in an internal buffer form which it can then be retrieved by the AOCS computer through a data acquisition operation.

5.3 Operation Model

The following types of operations can be performed upon unit objects:

- *Data Acquisition*

  Data are transferred from the external unit to their destination buffer in the data pool

- *Data Send*

  Data are transferred from a source buffer in the data pool to an external unit

- *Synchronization*

  The AOCS computer sends a synchronization command to the unit

- *Command Send*

  The AOCS computer sends a command to the unit to perform one of the following:

  – change in the current operational mode
  – unit reset
  – interruption of on-going bus transaction
  – unit initialization
  – initiate self-test

The above operations are executed in one or two phases:

- *Bus Transaction Phase*

  A *transaction* is an actual data or command exchange taking place on the physical link between the external unit and the AOCS computer.

- *Refresh Phase*

  A *refresh operation* cover the translation between raw and AOCS data levels, and hence a transfer between hardware buffers and destination / source buffers.

The refresh phase only applies to operations that result in an exchange of data (as opposed to commands) between the external unit and the AOCS computer.

The *order* in which the two phases are executed depends on the operation type. For data acquisition operations, the bus transaction must be executed first and followed by the refresh phase. For the data send operations, the opposite order applies.

The AOCS unit object exposes dedicated methods to perform each phase of each operation as shown in the table below. As explained in the next section, AOCS units present two interfaces to the rest of the AOCS software: the `AocsUnitFunctional` and the `AocsUnitHousekeeping` interface. The operations listed at the beginning of this section may refer either to the acquisition or sending of housekeeping or of functional data. Where two methods are presented for the same operation-phase pair in the table, one applies to the functional and the other to the housekeeping interface of AOCS units.

	Bus Transaction Phase	**Refresh Phase**
Data Acquisition Operation	`AcquireHousekeepingData` `acquireFunctionalData`	`refreshHousekeepingIn` `refreshFunctionalIn`
Data Send Operation	`sendHousekeepingData` `sendFunctionalData`	`refreshHousekeepingOut` `refreshFunctionalOut`

Synchronization Operation	`SynchronizeHousekeeping` `synchronizeFunctional`	`phase not applicable`
Command Send Operation	`see interface desc.`	`phase not applicable`

Calls to the bus transaction methods result in transactions being *initiated* on the communication link with the external unit. The transmission may take some time to complete. In some implementations, transaction methods may be non-blocking: they initiate the transaction and immediately return. Separate methods are provided to check whether the initiated transaction has been completed. Note that no call-back mechanism is foreseen whereby clients are directly notified of the termination of a transaction.

Refresh actions are always executed, regardless of whether new data have arrived since the previous refresh. In a future implementation, they might only be actually executed if they can lead to an update of the target buffer. If, for instance, no new data were received in the input buffer since the last time a `refreshIn` method was called, then a new call to the `refreshIn` method will return without performing any action. This would improve efficiency as it would avoid duplicate data conversion processes but would require associating and maintaining time-tags to the hardware buffers.

Refresh operations are carried out by control channel objects that are called *data converters* because they convert the unit data to and from raw data level. The data converter may include bias and scale factor corrections, coordinate frame transformation, pre-filtering, and any other operation that may have to be performed to the incoming or outgoing unit data. The AOCS unit data sees the data converter only through the abstract control channel interface and therefore need not be concerned about the exact nature of the operations performed by it.

As an example of a full data transfer, consider a data acquisition transfer cycle. This could be articulated over the following steps:

- Call `dataAcquire` to initiate the bus transaction to acquire data from the physical unit
- Check that the acquisition process is finished through method `isTransactionFinished`
- Call `refreshIn` to convert acquired data to AOCS level and transfer them to data pool.

At the end of this three-step process, the newly acquired data are located in a data pool in AOCS format and referred to the satellite reference frame. From the data pool, they are accessible to all other objects in the AOCS software.

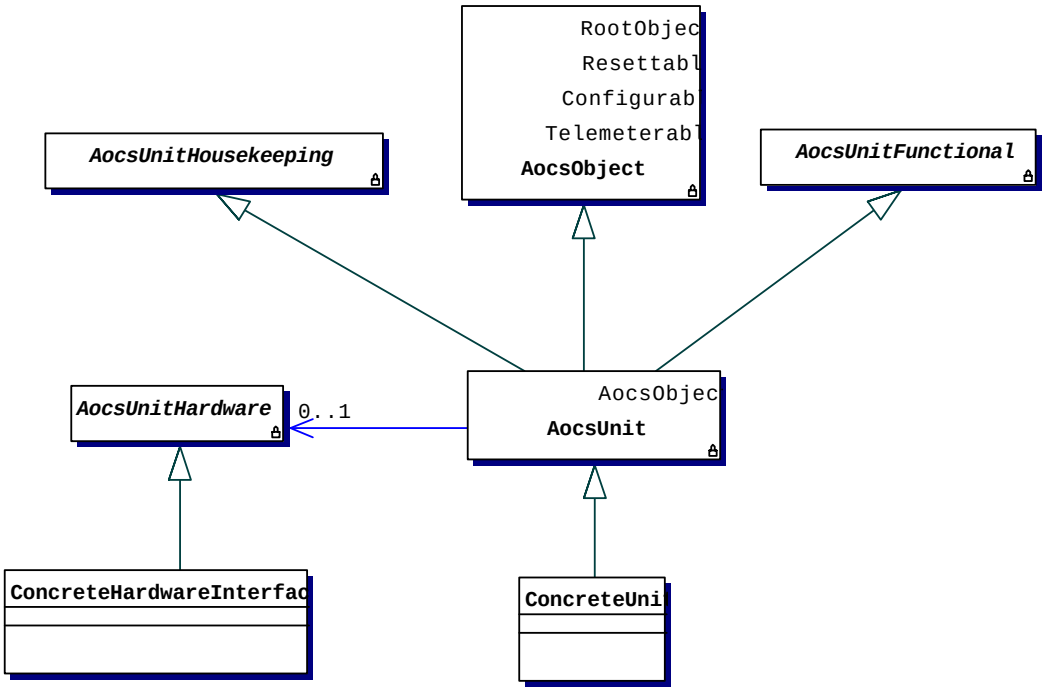
5.4 Model Limitations

The unit model proposed here treats all the incoming and outgoing data associated to a certain sensor as a single data block. The interfaces through which AOCS unit objects are seen (`AocsUnitFunctional` and `AocsUnitHousekeeping`) expose methods to acquire and to send data which act on the incoming and outgoing data blocks as a single unit. There is no way to command the acquisition or the sending of individual data items. Thus, this unit model is only suitable for real units which take as inputs or generate as outputs data that are logically related and that are to be provided or generated at approximately the same time.

5.5 AOCS Unit Class Structure

The internal structure of units varies enormously from unit to unit. Hence, no components encapsulating generic unit features can be provided by the framelet. The unit framelet thus consists exclusively of abstract classes. The framelet exports two interfaces – `AocsUnitFunctional` and `AocsUnitHousekeeping` – and an abstract class – `AocsUnit` – that implements the interfaces. Additionally, the framelet defines an interface – `AocsUnitHardware` – that is used only internally to the framelet.

The mutual relationships among these constructs are shown in the figure:

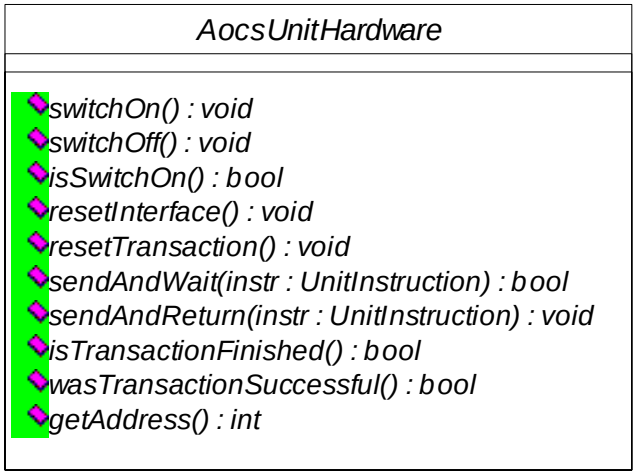


Concrete units *as seen by the rest of the AOCS software* are represented by instances of concrete classes derived from `AocsUnit`. Concrete unit objects delegate low level operations to objects of class `AocsUnitHardware`. `AocsUnitHardware` objects encapsulate the direct exchanges with the hardware. They are normally not visible outside the framelet. Several AOCS units may share access to the same AOCS unit hardware object.

Like most non-trivial objects in the AOCS software, unit objects are ultimately derived from class `AocsObject`.

6 HARDWARE UNIT OBJECTS

Direct interaction with the hardware – the AOCS bus linking the AOCS computer to the AOCS units – is delegated by AOCS unit objects to *hardware unit* objects. Hardware unit objects implement the interface `AocsUnitHardware`:



In general, a hardware unit object is associated to each external unit. In many cases, however, several units may share the same hardware unit object. Thus, for instance, in the case of a MACS-based AOCS, communication with the external units is through a MACS bus controller and a single hardware unit object – encapsulating the interface to the MACS controller – is shared by all unit objects.

The semantics of the methods defined by interface `AocsUnitHardware` are summarized in the table:

`switchOn(), switchOff(), isSwitchedOn()`
Control and check the unit's power status.
`sendInstructionAndReturn(UnitInstruction instr)`
Initiate the bus transaction encapsulated in the argument `instr` and return without waiting for the bus transaction to be completed ("non-blocking send").
`sendInstructionAndWait(UnitInstruction instr)`

	Initiate the bus transaction encapsulated in the argument `instr` and wait for its completion before returning ("blocking send").
`isTransactionFinished()`	
	Returns `true` if the last transaction has completed and returns false if the transaction is still under way.
`wasTransactionSuccessful()`	
	Returns `true` if the last completed transaction terminated without reporting any errors.
`resetTransaction()`	
	Reset any on-going bus transaction.
`resetInterface()`	
	Command a hardware reset to the interface with the external units.

Essentially, hardware unit objects serve as vehicles to send instructions to real units without making any commitment to the specific type of link connecting the units to the AOCS computer. All higher level functions – management of health status, data conversions, failure reporting, etc – are left to AOCS unit objects. Hardware units therefore encapsulate the mechanism used to put outgoing data and commands on the bus and to retrieve incoming data from the bus. The data and instruction to be put on or retrieved from the bus are stored in the `unitInstruction` structure described in the next sub-section.

6.1 The `UnitInstruction` Structure

The `UnitInstruction` structure is defined as follows:

```
struct UnitInstruction {
     unsigned short priority;
     unsigned short address;
     unsigned short subAddress;
     unsigned short command;
     unsigned short* data;
     unsigned short nDataWords;
}
```

This structure provides fields for the definition of the information required to construct a generic instruction for the hardware interface controlling the exchanges between the AOCS computer and an external unit. The actual construction of the instruction is done by the `AocsUnitHardware` object that takes the data in `UnitInstruction` and assembles them in the manner required to control the unit hardware interface.

6.2 Abstraction Levels

It is important to appreciate that this framelet introduces two levels of abstractions to handle external units. The abstract class `AocsUnit` discussed in the next section provides an abstract interface through which individual units can be managed by the AOCS software. The `AocsUnitHardware` interface and the `UnitInstruction` structure are instead intended to provided generic handles through which the hardware interface to external units can be managed by the `AocsUnit` objects. Since such hardware interfaces come in many different kinds, it is possible that in some cases the abstract operations declared by `AocsUnitHardware` or the abstract fields offered by `UnitInstruction` may not be adequate. In that case, application-specific interfaces would have to be defined. However, the existence of the second layer of abstraction – the `AocsUnit` interface – would mean that such change would have no impact on the AOCS framework and only a minimal impact on the AOCS software.

6.3 The `MacsTcController` Component

The AOCS framework offers one default component implementing the `AocsUnitHardware` interface. Component `MacsTcController` encapsulates the interface to a MACS telecom controller (see RD3).

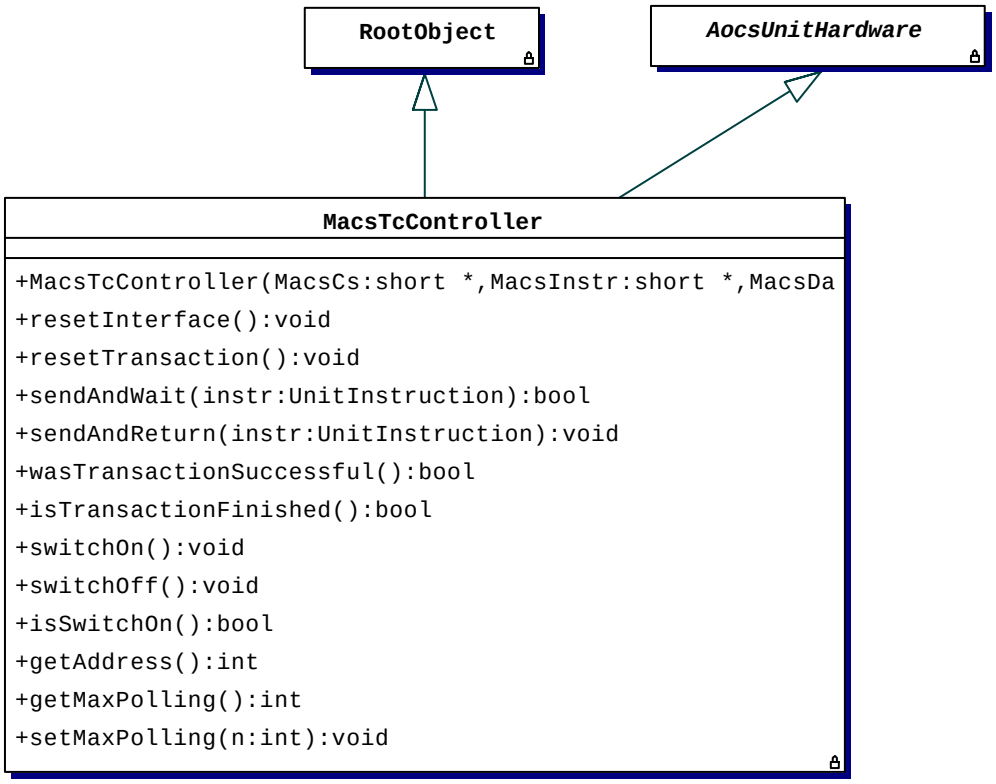
The component assumes that the communication with the MACS controller is via memory-mapped I/O as described in detail in RD4.

The `MacsTcController` component maps the `UnitInstruction` fields to the instruction fields required by the MACS TC protocol according to the following table:

`UnitInstruction` Field	MACS TC Instruction Field
`priority`	extension
`subAddress`	subaddress to which instruction is sent
`address`	address of receiving unit

`command`	command
`data`	address of data word associated to instruction
`nDataWords`	not used by the MACS TC protocol

The class definition of component `MacsTcController` is:



Thus, `MacsTcController` adds only two operations `getMaxPolling` and `setMaxPolling` to those defined by the `AocsUnitHardware` interface. These operations are used to set and get the `maxPolling` parameter. Component `MacsTcController` uses a polling mechanism to implement method `sendInstructionAndWait`: it puts the instruction on the bus and then loops until the "instruction sent" bit in the MACS status register indicates that the instruction, and any data words associated to it, were put on the bus. Parameter `maxPolling` defines the maximum number of cycles in the waiting loop before a failure is declared to have occurred.

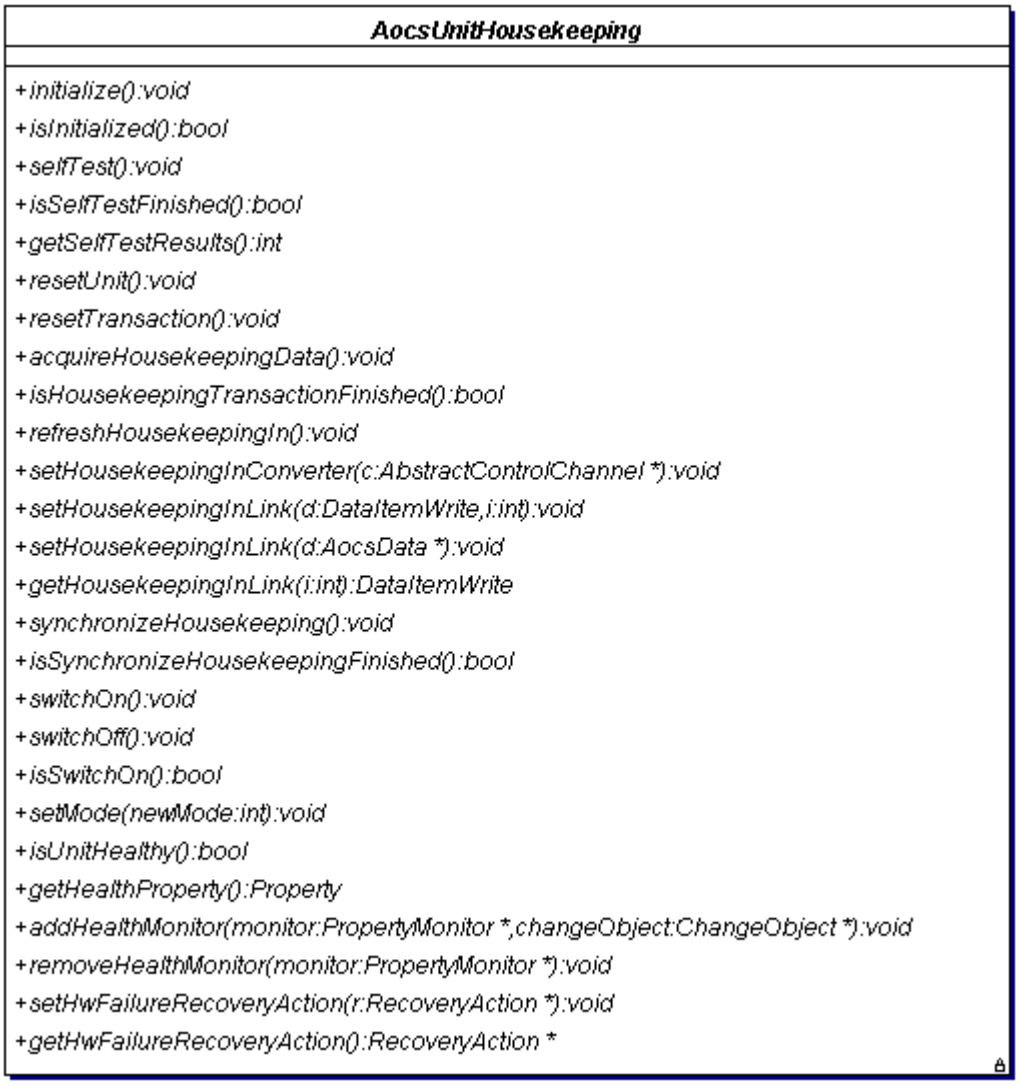
The class constructor takes three parameters that define the addresses of the registers where the MACS control status, instruction and data word are located. See RD4 for their layout.

7 AOCS UNIT OBJECTS

External units are represented inside the AOCS software by proxy objects called *AOCS unit objects.* AOCS unit objects are instances of class `AocsUnit`. Their complete class diagram is shown in the figure of section 5.5. Class `AocsUnit` adds no operations to those it inherits from interfaces `AocsUnitFunctional` and `AocsUnitHousekeeping` which are discussed in the next two subsections.

7.1 The `AocsUnitHousekeeping` Interface

The `AocsUnitHousekeeping` interface is defined as follows:



The semantics of the methods defined by this interface are summarized in the table below:

`selfTest(), isSelfTestFinished(), getSelfTestResults()`
`sefTest` Initiate a self-test and returns without waiting for the self-test to be terminated. Method `isSelfTestFinished`() returns true when the self test has been completed and method `getSelfTestResults` results an integer representing the outcome of the self-test. The interpretation of this return value is unit-specific.
`switchOn(), switchOff(), isSwitchedOn()`
Control and retrieve the power status of the unit.
`acquireHousekeepingData(), isHousekeepingTransactionFinished()`
A call to `acquireHousekeepingData` initiates the bus transaction(s) for the acquisition of housekeeping data from the unit. This method may be non-blocking and the completion status of the acquisition can be checked by calling `isHousekeepingAcquisitionFinished`().
`resetTransaction()`
Resets any on-going transactions (of any type).
`refreshHousekeepingIn()`
Housekeeping data are acquired at raw data level and placed in a hardware buffer internal to the AOCS unit. A call to this method causes them to be converted to AOCS data level and transferred to a data pool location.
`setHousekeepingInConverter(AbstractControlChannel* cc)`
Data conversion from raw data level to AOCS data level, and any other processing on the raw data (eg. bias correction), is done by a control channel object. This method allows the converter control channel to be loaded. Note that a unit of housekeeping data may consist of several individual data items (see section 5.4). Thus, the control channel will in general be a multi-input-multi-output control channel. The correct number of inputs and outputs is checked when the control channel is loaded. An incorrect number of inputs and/or outputs represents a configuration error.
`setHousekeepingInLink(DataItemWrite diw, int i)`
This method specifies the destination of the housekeeping data in the data pool. It establishes a link with the destination buffer. More specifically, it ensures that the j-th housekeeping datum collected by the unit is written to the write data item `diw`. Note

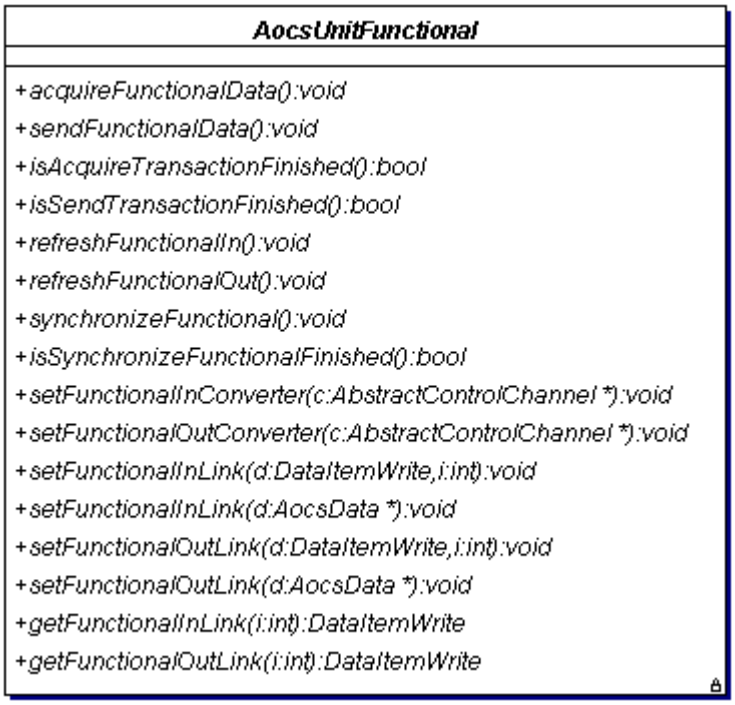
that links to the input buffers are hard-coded in the AOCS unit object.
`synchronizeHousekeeping(), isSynchronizeHousekeepingFinished()`
The first method initiates a synchronization operation for the housekeeping data and the second one checks whether the underlying bus transaction has been completed.
`setMode(int i)`
Units can have different operational modes. This method sets the unit in the i-th operational mode.
`isHeathy(), getHealthProperty()`
A health status can be associated to units. The health status is checked by calling `isHealthy`. The health status is treated as a property and method `getHealthProperty` returns the associated property object.
`addHealthMonitor(Monitor* m, ChangeObject c), removeHealthMonitor(Monitor* m)`
The health status is a bound property that can be subjected to monitoring through change notification. These two methods allow property monitors to register and unregister their interest in certain types of changes in the health status.
`initialize`(), `isInitialized()`
AOCS Units often have to perform some kind of initialization procedure after being switched on. A call to `initialize`() causes this initialization procedure to be carried out. The initialization procedure may involve bus transactions that take some time. The method may be non-blocking and the completion status of the initialization procedure can be checked by calling method `isInitialized`.
`resetUnit()`
Reset the unit. Note that this is different from the generic reset method that applies to all AOCS objects: the latter resets the software object, the former issues commands to reset an external unit.

Typically, the implementation of the `acquireHousekeepingData` and other operations involving direct interaction with the external unit are delegated to a hardware unit object. The housekeeping data acquired through a call to `acquireHousekeepingData` are automatically placed in the appropriate hardware buffer. The hardware buffers are internal to the unit proxy object and the link to them is hardcoded in the unit object.

Note that, as discussed in section 5.4, housekeeping data are acquired as a single block of data. The call to `acquireHousekeepingData` may be translated into several low level acquisition instructions, one for each individual data item in the unit housekeeping data block. From the point of view of the rest of the AOCS software, however, only one acquisition operation occurs.

7.2 The `AocsUnitFunctional` Interface

The `AocsUnitFunctional` interface is defined as follows:



The semantics of the methods defined by this interface are summarized in the table below:

`acquireFunctionalData(), isAcquireTransactionFinished()`
A call to `acquireFunctionalData` initiates the bus transaction(s) for the acquisition of functional data from the unit. This method may be non-blocking and the completion status of the acquisition can be checked by calling `isAcquireTransactionFinished`().
`refreshFunctionalIn()`
Functional data are acquired at raw data level and placed in a hardware buffer

internal to the AOCS unit. A call to this method causes them to be converted to AOCS data level and transferred to a data pool location.
`setFunctionalInConverter(AbstractControlChannel* cc)`
Data conversion from raw data level to AOCS data level, and any other processing on the raw data (eg. bias correction), is done by a control channel object. This method allows the converter control channel to be loaded. Note that a unit of functional data may consist of several individual data items (see section 5.4). Thus, the control channel will in general be a multi-input-multi-output control channel. The correct number of inputs and outputs is checked when the control channel is loaded. An incorrect number of inputs and/or outputs represents a configuration error.
`setFunctionalInLink(diw, i), getFunctionalInLink`
The setter method specifies the destination of the functional data in the data pool. It establishes a link with the destination buffer. More specifically, it ensures that the j-the functional datum collected by the unit is written to the write data item `diw`. Note that links to the input buffers are hard-coded in the AOCS unit object. The getter method returns the data item write object associated to the i-th destination buffer.
`sendFunctionalData(), isSendTransactionFinished()`
A call to `sendFunctionalData` initiates the bus transaction(s) to send the functional data to the unit. This method may be non-blocking and the completion status of the acquisition can be checked by calling `isSendTransactionFinished()`.
`refreshFunctionalOut()`
Functional data to sent to the external unit are taken from hardware buffer internal to the AOCS unit. A call to this method causes the content of the hardware buffer to be updated with the latest data from the source buffers in the data pool.
`setFunctionalOutConverter(AbstractControlChannel* cc)`
Data conversion from AOCS data level (in the source buffers) to raw data level (in the hardware buffer) is done by a control channel object. This method allows the converter control channel for outgoing data to be loaded. Note that a unit of functional data may consist of several individual data items (see section 5.4). Thus, the control channel will in general be a multi-input-multi-output control channel. The correct number of inputs and outputs is checked when the control channel is loaded. An incorrect number of inputs and/or outputs represents a configuration error.

`setFunctionalOutLink(dir, i), getFunctionalOutLink(i)`
This method specifies the source in the data pool of the data to be sent to the external unit. It establishes a link with the source buffer. More specifically, it ensures that the j-th functional datum to be sent out by the unit is taken from the read data item `dir`. Note that links to the input buffers are hard-coded in the AOCS unit object. The getter method returns the data item write object associated to the i-th source buffer.
`synchronizeFunctional(), isSynchronizeFunctionalFinished()`
The first method initiates a synchronization operation for the functional data and the second one checks whether the underlying bus transaction has been completed.

The implementation of most of the methods in the table is delegated by the AOCS unit objects to its associated hardware unit object.

Note that, as discussed in section 5.4, functional data are acquired and sent as a single block of data. Hence calls to `acquireFunctionalData` and `sendFunctionalData` may be translated into several low level acquisition/send instructions, one for each individual data item in the unit functional data block. From the point of view of the rest of the AOCS software, however, only one acquisition operation occurs.

7.3 Error Handling

Neither of the two AOCS unit interfaces expose operations to check the success of failure of a transaction with an external unit. Error checking is done internally to the unit object and invisibly to its user. Low level errors can be detected by calling method `wasTransactionSuccessful` on the hardware unit object. Such errors are reported as failure events.

When a unit object has detected an error in a transaction with incoming data, it may, depending on the type of error, refuse to perform a `refreshIn` operation. In other words, the unit object may refuse to update the source buffer with data that are known or suspected to be erroneous. Refusal of service is done invisibly to the caller of `refreshIn` who has no way of knowing whether its refresh request has been carried out or not[1]. This approach is in line with the inter-component communication philosophy that decouples production of data

[1] Unless, of course, it were to check whether any bus failure events have been generated in the failure event repository.

from their consumption and allows components to exchange data only through shareable data areas

7.4 Non-Nominal Transactions

The AOCS unit interfaces presented above do not give unfettered access to the underlying hardware units. Access is restricted to the forms foreseen by the unit mode described in section 5. The main reason for this restriction is the need to define a *generic* interface that can fit *all* types of units currently in use and the desirability of allowing uniform treatment of real and fictitious units.

In most cases, the AOCS unit interface will be sufficient to control the external units. When a unit needs finer control, however, this can be provided by giving direct access to its hardware unit object. In practice this can be done by adding a method with the following signature:

```
AocsUnitHardware* getAocsHardwareUnit()
```

to the concrete class derived from class `AocsUnit`.

The concrete unit holds a reference to the hardware unit. Normally, this reference is not accessible from the outside but where required the concrete unit class can be endowed with a method to return it. Access to the hardware unit gives access to its `sendInstruction` operations that can be used to operate directly on the real unit.

An alternative implementation might have one or both of the AOCS unit interfaces expose a generic `getAocsUnitHardware` method that returns the hardware unit associated to the unit. This was not done because direct operations on the hardware unit should, whenever possible, be avoided and will be necessary in only rare instances

7.5 Split Units

One of the main limitations of the unit model adopted here is discussed in section 5.4: units can only acquire or send one single block of data in atomic operations. Only one `send` and only one `acquire` methods are provided by the `AocsUnit` interface and the methods have no parameters so that there is no way to specify *which* unit data should be sent or acquired. It is always assumed that to each unit a single block of send and a single block of acquire data are associated. This section outlines one way in which this limitation could be overcome.

Consider a hypothetical unit that generates both angular velocity and attitude measurements and suppose that the two measurements are generated (or required) at different rates. This unit clearly does not fit the unit model proposed above since its outputs cannot be treated as

an atomic data block. One way to address this problem is to create *two* `AocsUnit` objects to model the unit in the AOCS software.

One object will be responsible for the collection and management of the angular velocity measurements while the other will be responsible for the attitude measurements. The original unit is now seen within the AOCS software as two simpler units. It will be said that the unit has been *split* into two units.

`AocsUnit` objects interact with external units through a hardware unit object that encapsulates the low-level hardware interface to the unit. Normally, to each AOCS unit object, there corresponds one dedicated hardware object. In the case of split units however, all `AocsUnit` objects split from the same real unit are linked to the same `HardwareUnit` object. It then becomes necessary for the `AocsUnit` objects to coordinate their behaviour and changes in their internal state to ensure that they present a consistent picture of the unique real unit that is behind them. Thus, for instance, if one `AocsUnit` object issues a `resetTransaction` command, other `AocsUnit` objects need to be notified since their on-going transactions will also be affected. Similarly, if one `AocsUnit` object changes the unit health status, the other `AocsUnit` objects again need to be notified.

This type of coordination could be naturally implemented using the property monitoring mechanism. This implementation, however, is not offered as a default by the framework since split units are not required to model units currently in use in AOCS systems and therefore this facility is outside the boundaries of the AOCS framework.

8 THE AOCSUNIT CLASS

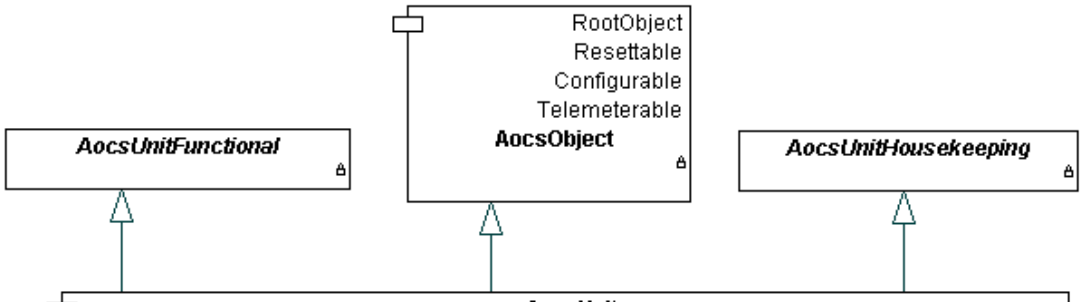
Class AocsUnit is defined in the figure in the next page.

Thus, AocsUnit only adds a constructor and two sets of operations to those it inherits as shown in the table below:

AocsUnit(nHkInpBuf,nHkDdBuf,nFcInpBuf,nFcDdBuf,nFcOutBuf,nFcDsBuf)
Constructor specifying the number of functional (Fc) and housekeeping (Hk) hardware input (Inp) buffers and destination buffers (Dd) and source buffers. This allows the constructor to allocate the memory for its internal buffers (including the hardware buffers)
setFcHwInpBuff(),getFcHwInpBuff()
The hardware input buffers are normally set by the internal AocsUnit logic in response to the arrival of data from the external unit and are not accessible from outside the object. However, AOCS units objects are often tested without hardware connections to the real external units These methods allow the hardware input buffers to be set and read during such testing phases.
getAocsUnitHardware(AocsUnitHardware* u), setAocsUnitHardware()
Getter and setter method for the unit hardware object associated to the AOCS unit object.

Class AocsUnit is not abstract because it provides default implementations for all the methods it inherits from the AOCS unit interfaces. In many cases, trivial implementations are provided to facilitate the construction of the concrete AOCS unit subclasses which, in most cases, will only need to implement a small subset of the methods declared by AocsUnitFunctional and AocsUnitHousekeeping. Class AocsUnit provides non-trivial implementation for the operations to manage the data flow between the hardware buffers and the destination/source buffers. These operations can in general be taken over by concrete unit subclasses since the management of this data flow will usually not depend from the specific unit model.

The next section describes a few concrete AOCS unit objects that are provided as default components by the AOCS framework.



AocsUnit

#localReset():void
#localResetConfiguration():void
+AocsUnit(nHkInpBuf:int,nHkDdBuf:int,nFcInpBuf:int,nFcDdBuf:int,nFcOutBuf:int,nFcDsBuf:int)
+initialize():void
+isInitialized():bool
+selfTest():void
+isSelfTestFinished():bool
+resetUnit():void
+resetTransaction():void
+acquireHousekeepingData():void
+isHousekeepingTransactionFinished():bool
+acquireFunctionalData():void
+sendFunctionalData():void
+isAcquireTransactionFinished():bool
+isSendTransactionFinished():bool
+refreshHousekeepingIn():void
+refreshFunctionalIn():void
+refreshFunctionalOut():void
+setHousekeepingInLink(d:DataItemWrite,i:int):void
+setFunctionalInLink(d:DataItemWrite,i:int):void
+setFunctionalOutLink(d:DataItemWrite,i:int):void
+setHousekeepingInLink(d:AocsData *):void
+setFunctionalInLink(d:AocsData *):void
+setFunctionalOutLink(d:AocsData *):void
+getHousekeepingInLink(i:int):DataItemWrite
+getFunctionalInLink(i:int):DataItemWrite
+getFunctionalOutLink(i:int):DataItemWrite
+synchronizeHousekeeping():void
+isSynchronizeHousekeepingFinished():bool
+synchronizeFunctional():void
+isSynchronizeFunctionalFinished():bool
+switchOn():void
+switchOff():void
+isSwitchOn():bool
+isUnitHealthy():bool
+setFcHwInpBuff(d:unsigned short,i:int):void
+addHealthMonitor(monitor:PropertyMonitor *,changeObject:ChangeObject *):void
+removeHealthMonitor(monitor:PropertyMonitor *):void
+reset():void
+resetConfiguration():void
+isConfigured():bool
+writeToTelemetry(stream:TelemetryStream *):void

selfTestResults:int
housekeepingInConverter:AbstractControlChannel *
functionalInConverter:AbstractControlChannel *
functionalOutConverter:AbstractControlChannel *
mode:int
fcHwOutBuff:unsigned short[]

8.1 The Telemetry Interface

AOCS objects are telemetry objects because they inherit from `AocsObject` the `telemeterable` interface.

The data sent to the telemetry stream by an `AocsUnit` object in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	hardware buffers + health status
Long	normal TM + instance ID of hardware unit object
Debug	long TM + links to source and destination buffers + ID of converters

8.2 The Reset and Configurable Interface

AOCS objects inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

AOCS unit defines a class-specific `Reset` method that clears all hardware buffers and resets all converters.

AOCS unit defines a class-specific `resetConfiguration` method that unloads the converter objects and the AOCS unit hardware object and clears all links to the source and destination buffers.

AOCS unit defines a class-specific method `isConfigured` that returns true if: converter objects have been loaded; links have been set up to all source and destination buffers; an AOCS unit object has been loaded.

9 CONCRETE AOCS UNIT OBJECTS

The primary constructs exported by the unit framelet are the abstract AOCS unit interfaces. In its mature state, however, the framelet will also provide a number of default implementations of these interfaces serving as proxies for commonly used AOCS units. The prototype framework only offers four such default components representing, respectively, a fine sun sensor, a single-axis gyro, a sun acquisition and propulsion electronics, and a reaction wheel. In all cases, the units will assume a MACS-based interface (telecom protocol) and they will only acquire or send functional data. These default units were developed for the AOCS prototype testing campaign (see RD5).

9.1 The `FssPrototype` Unit

The `FssPrototype` class is a concrete AOCS unit class (see UML diagram of section 5.5) that encapsulates a simple two-axis fine sun sensor (FSS).

The operations specific to this class are described in the table:

`FssPrototype(address, subAddressFssX, subAddressFssY)`	
	Constructor that defines the MACS address of the FSS unit and the MACS subaddresses from which the FSS read-outs representing the sun position on the sensor X and Y axes are retrieved.
`initialize()`	
	Initialize the FSS unit by sending it a MACS RC instruction to initialize its MACS controller.
`synchronizeFunctional()`	
	Send a MACS broadcast instruction that causes the FSS measurements tobe latched in the internal register from which they are acquired with an `acquireFunctional` operation. Note that, as a broadcast, the MACS instruction associated to this operation is received by *all* MACS units on the bus.
`acquireFunctional()`	
	Send two MACS TI instructions in sequence to acquire the channel X and channel Y read-outs of the FSS. The operation is implemented to return only after the bus transaction has been concluded and the FSS data have been written to the hardware buffers.

`setMacsFailureRecoveryAction(),getMacsFailureRecoveryAction`
If a MACS bus failure is reported by the hardware unit object associated to the unit, a failure event is generated. These are the getter and setter methods for the recovery action associated to this failure event.

9.2 The `GyrPrototype` Unit

The `GyrPrototype` class is a concrete AOCS unit class (see UML diagram of section 5.5) that encapsulates a simple single-axis gyro sensor (GYR).

The operations specific to this class are described in the table:

`GyrPrototype(address, rateAddress)`
Constructor that defines the MACS address of the GYR unit and the MACS subaddress from which the GYR read-outs representing the spacecraft angular rate around the gyro's sensing axis is retrieved.
`initialize()`
Initialize the GYR unit by sending it a MACS RC instruction to initialize its MACS controller.
`acquireFunctional()`
Send one MACS TI instruction to subaddres `rateAddress` to acquire the rate measurement. The operation is implemented to return only after the bus transaction has been concluded and the GYR data has been written to the hardware buffers.
`setMacsFailureRecoveryAction(),getMacsFailureRecoveryAction`
If a MACS bus failure is reported by the hardware unit object associated to the unit, a failure event is generated. These are the getter and setter methods for the recovery action associated to this failure event.

9.3 The `RwPrototype` Unit

The `RwPrototype` class class is a concrete AOCS unit class (see UML diagram of section 5.5) that encapsulates a simple reaction wheel actuator (RW).

The operations specific to this class are described in the table:

`RwPrototype(address, subAddressSpeed, subAddressTorque)`	
	Constructor that defines the MACS address of the RW unit and the MACS subaddress from which the wheel speed is retrieved and to which wheel torque commands are sent.
`initialize()`	
	Initialize the RW unit by sending it a MACS RC instruction to initialize its MACS controller.
`acquireFunctional()`	
	Send one MACS TI instruction to subaddres `subAddressSpeed` to acquire the wheel speed measurement. The operation is implemented to return only after the bus transaction has been concluded and the speed data has been written to the hardware buffers.
`sendFunctional()`	
	Send one MACS RD instruction to subaddress `subAddressTorque` to send the wheel torque command. The operation is implemented to return only after the bus transaction has been concluded and the torque data has been put on the MACS bus.
`setMacsFailureRecoveryAction(),getMacsFailureRecoveryAction`	
	If a MACS bus failure is reported by the hardware unit object associated to the unit, a failure event is generated. These are the getter and setter methods for the recovery action associated to this failure event.

9.4 The `SapPrototype` Unit

The `SapPrototype` class is a concrete AOCS unit class (see UML diagram of section 5.5) that encapsulates a simple Sun Acquistion and Propulsion (SAP) electronics unit. The SAP unit controls a set of 3 sun acquisition sensors (SAS) and 6 thrusters (THU).

A SAP unit provides 12 SAS measurements corresponding to channel X and channel Y measurements for each SAS.

A SAP unit can receive 12 commands corresponding to the delay-time and on-time command for each thruster.

The operations specific to this class are described in the table:

`SapPrototype(address, sasBaseSubaddress, thuBaseSubaddress)`	
	Constructor that defines the MACS address of the SAP unit and the MACS subaddress from which the SAS measurements are retrieved and to which THU commands are sent. Subaddress (`sasBaseSubaddress+2*i`) provides the channel X measurement for the i-th SAS. Subaddress (`sasBaseSubaddress+2*i+1`) provides the channel Y measurement for the i-th SAS. Subaddress (`thuBaseSubaddress+2*i`) receives the delay-time command for the i-th THU. Subaddress (`thuBaseSubaddress+2*i+1`) receives the on-time command for the i-th THU.
`initialize()`	
	Initialize the SAP unit by sending it a MACS RC instruction to initialize its MACS controller.
`acquireFunctional()`	
	Send six MACS TI instruction to subaddress `sasBaseSubaddress` to (`sasBaseSubaddress+5`) to acquire the SAS measurements. The operation is implemented to return only after the bus transaction has been concluded and the SAS data have been written to the hardware buffers.
`sendFunctional()`	
	Send twelve MACS RD instructions to subaddress `thuBaseSubaddress` to (`thuBaseSubaddress+11`)to send the THU delay-time and on-time commands. The operation is implemented to return only after the bus transaction has been concluded and the THU data have been put on the MACS bus.
`setMacsFailureRecoveryAction(),getMacsFailureRecoveryAction`	
	If a MACS bus failure is reported by the hardware unit object associated to the unit, a failure event is generated. These are the getter and setter methods for the recovery action associated to this failure event.

9.5 The Telemetry Interface

The concrete unit objects described in this section are telemetry objects because they inherit the `telemeterable` interface. The data they send to the telemetry stream in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	none
Long	instance ID of MACS failure recovery action
Debug	long TM + MACS addresses and subaddresses

9.6 The Reset and Configurable Interface

The concrete unit objects described in this section inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

No class-specific `Reset` method is defined.

A class-specific `resetConfiguration` method is defined that unloads the MACS failure recovery action.

A class-specific method `isConfigured` is defined that returns true if the MACS failure recovery action has been loaded.

9.7 Unit Data Converters

AOCS unit objects use control channel objects to perform the data conversions between raw data level and AOCS data level. Three converter control channels are defined specifically for the prototype AOCS unit described in this section:

- `RawBusDataConverterIn`

  Performs data conversion from raw to AOCS data level for incoming data for a unit with m input channels. The following format is assumed for the raw bus data from each channel:

  – bits 0-14 : binary representation of absolute value of datum
  – bit 15 : sign bit (datum is positive if bit is equal to 1)

The number of channels m and the resolution level for the raw bus datum are passed as constructor parameters.

- `RawBusDataConverterOut`

  Performs data conversion from AOCS to raw data level for outcoming data for a unit with m output channels. The following format is assumed for the raw bus data from each channel:

  – bits 0-14 : binary representation of absolute value of datum
  – bit 15 : sign bit (datum is positive if bit is equal to 1)

  The number of channels m and the resolution level for the raw bus datum are passed as constructor parameters.

- `BiasScalingCompensator`

  Performs bias and scaling factor compensation according to the following formula:

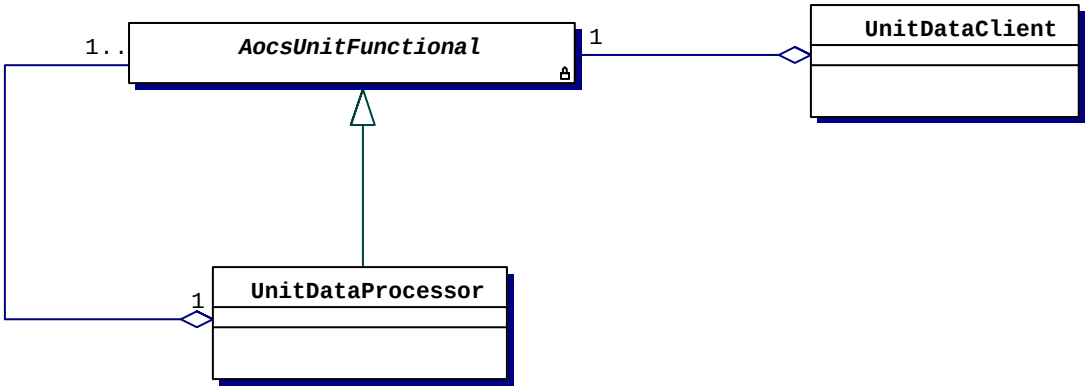
  ```
  output = bias + scalingFactor * input
  ```

10 FICTITIOUS UNIT DESIGN PATTERN

The fictitious unit design pattern is introduced to address the problem of combining components that process unit data without impacting the final users of the unit data.

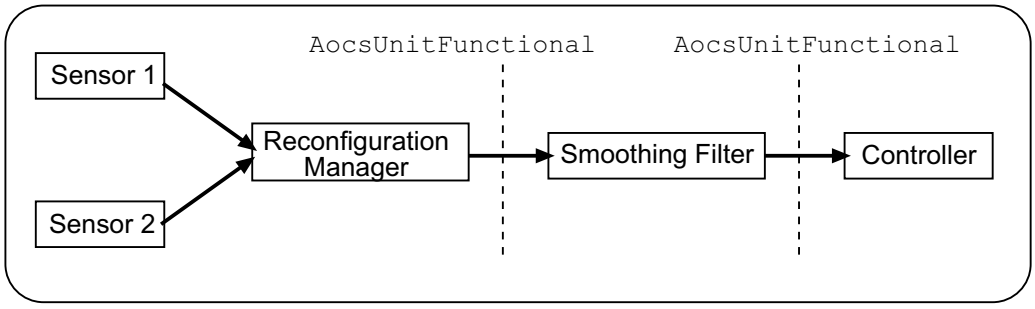
The design pattern is based on the concept of *fictitious AOCS unit* that is defined as an object that implements the `AocsUnitFunctional` interface (see RD2 for some concrete examples). By implementing this interface, an object offers to its clients the same *functional* interface as an AOCS unit.

The fictitious unit pattern is illustrated by the following UML diagram:



The `UnitDataProcessor` is a concrete class that performs some kind of processing on the unit data. It is a fictitious AOCS unit because it implements interface `AocsUnitFunctional`. The unit data processor obtains the unit data from components that it sees as instances of type `AocsUnitFunctional`. These components may either true AOCS units or fictitious AOCS units. Interaction through the `AocsUnitFunctional` interface shields the unit data processor from having to know whether it is interacting with a real or a fictitious unit. `UnitDataClient` is the final user of the unit data. It too sees its source of unit data as an instance of type `AocsUnitFunctional` with the same advantages.

The fictitious data unit concept allows data processors to be easily combined without disrupting client's operation. An example of a combination of unit data processing elements is shown in the figure using informal notation:



The controller is the final user of the sensor data. Its operation is independent of how many filters and other processing elements are interposed between itself and the actual sensors.

The fictitious unit design pattern can also be seen as an instance of the composite pattern of RD1.

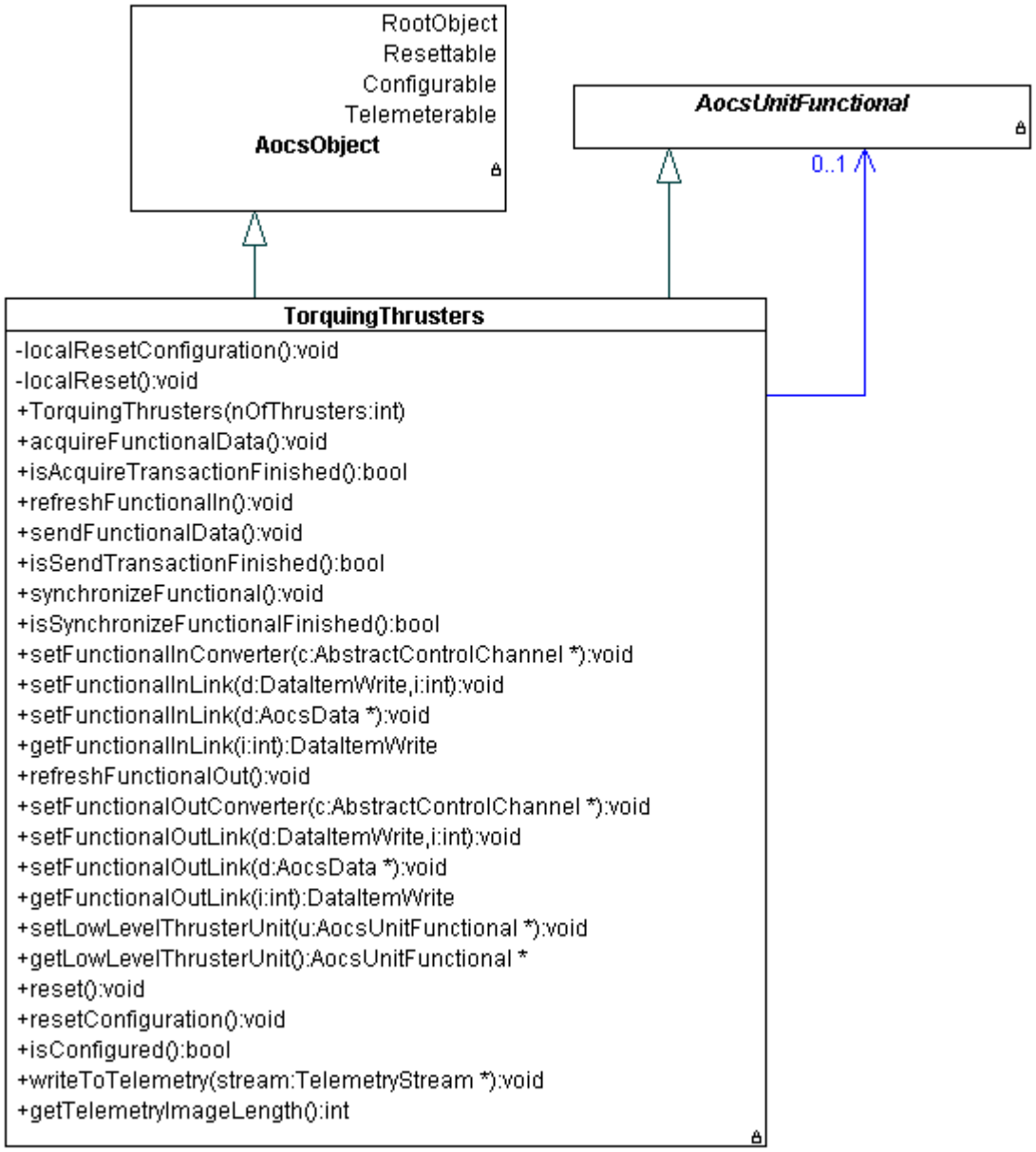
10.1 Recursion

The fictitious unit design pattern introduces recursion (as does its close relative the composite pattern). Calls to `AocsUnitFunctional` methods may be recursive and the maximum depth of recursion is given by the maximum number of elements that are linked together in a fictitious unit chain.

10.2 The `TorquingThrusters` Fictitious Unit

The AOCS framework prototype provides an example of fictitious unit with the `TorquingThrusters` class that is designed for use in the AOCS prototype (see RD5). The purpose of this class is to offer a high-level interface to the set of thrusters in the SAP prototype unit. The SAP thrusters are commanded through delay-time and on-time commands sent to the six individual thrusters. `TorquingThrusters` object instead allow the thruster set to be commanded directly with the torque requests around the spacecraft axes. They take care of converting these torque requests to on-time and delay-time requests for each thruster and do so invisibly to their client that can thus maintain the illusion of dealing with a set of ideal thrusters that apply torques directly around the spacecraft axes.

The class diagram for `TorquingThrusters` is:



The class adds only one method – `setLowLevelThrusterUnit` – to those it inherits from its parent classes. This new method is used to load the component implementing the low-level thruster unit. Note that this component is seen as an instance of type `AocsUnitFunctional` so that no assumptions are made about the specific type of AOCS unit that must be plugged in. This makes it possible to build chains of thruster command processing units.

10.3 The Reaction Wheel Set Fictitious Unit

Class `RwSet` is another example of fictitious unit provided by the prototype framework. The real units in this case are a set of four reaction wheels. Normally, only three out of the four wheel are used with the fourth one being kept as a redundant unit for use in case of failure. Class `RwSet` encapsulates the management of the redundancies among the four units and it gives the reaction wheel client the illusion of dealing with a fixed set of (non-redundant) reaction wheels.

This is an example of a more general situation where unit reconfiguration managers are implemented as fictitious units.

11 TRIGGER LISTS

A *trigger list* is an object that holds a list of references to `AocsUnitFunctional` or `AocsUnitHousekeeping` objects together with the type of operation – data acquisition, data sending or synchronization – that needs to be performed upon them. As discussed in the next section, a trigger list maintains a list of units upon which a certain operation – either a synchronization, or a data acquisition, or a data send – should be performed at the same time in the AOCS cycle. Trigger lists also provide operations to iterate through the items in the list.

Trigger lists are useful for the construction of unit trigger objects discussed in the next section.

Trigger lists are characterized by the following abstract interface:

The semantics of the methods defined by this interface is described in the table:

`addHousekeepingSynchronize(),removeHousekeepingSynchronize()`
Add and remove an item of type `AocsUnitHousekeeping` to which a synchronize

message should be sent.
`addHousekeepingAcquire(),removeHousekeepingAcquire()`
Add and remove an item of type `AocsUnitHousekeeping` from which housekeeping data should be acquired.
`addFunctionalSynchronize(),removeFunctionalSynchronize()`
Add and remove an item of type `AocsUnitFunctional` to which a synchronize message should be sent.
`addFunctionalAcquire(),removeFunctionalAcquire()`
Add and remove an item of type `AocsUnitFunctional` from which functional data should be acquired.
`addFunctionalSend(),removeFunctionalSend()`
Add and remove an item of type `AocsUnitFunctional` to which functional data should be sent.
`addInitShutDownUnit(),removeInitShutDownUnit ()`
Add and remove an item of type `AocsUnitHousekeeping` representing a unit upon which initialization and shut-down operations must be performed when the list is switched in/switched out at a mode transition (see section 12).
`firstHousekeepingSynchronize(), nextHousekeepingSynchronize(), isLastHousekeepingSynchronize()`
Iterator methods to go through the list of items of type `AocsUnitHousekeeping` to which synchronization messages should be sent.
`firstHousekeepingAcquire(), nextHousekeepingAcquire(), isLastHousekeepingAcquire()`
Iterator methods to go through the list of items of type `AocsUnitHousekeeping` from which housekeeping data should be acquired.
`firstFunctionalSynchronize(), nextFunctionalSynchronize(), isLastFunctionalSynchronize()`
Iterator methods to go through the list of items of type `AocsUnitFunctional` to which synchronization messages should be sent.
`firstFunctionalAcquire(), nextFunctionalAcquire(), isLastFunctionalAcquire()`

	Iterator methods to go through the list of items of type `AocsUnitFunctional` from which functional data should be acquired.
`firstFunctionalSend(), nextFunctionalSend(), isLastFunctionalSend()`	
	Iterator methods to go through the list of items of type `AocsUnitFunctional` fto which functional data should be sent.
`initialize()`	
	Operation to be called when the list is first switched in after an operational mode transition. It causes all the units associated to the list to be initialized (see section 12).
`shutdown()`	
	Operation to be called when the list is first switched out after an operational mode transition. It causes all the units associated to the list to be shutdown (see section 12).

Trigger lists are implemented using up to six object lists that hold the references to items of type `AocsUnitFunctional` and `AocsUnitHousekeeping` divided by type of operation – synchronize, data acquire, data send and initialize / shutdown – to be performed upon them.

The prototype AOCS framework offers two default implementation of interface `TriggerList`:

- `FullTriggerList`

  Full implementation of a trigger list allowing both kinds of items – `AocsUnitFunctional` and `AocsUnitHousekeeping` – and all three types of operations – synchronization, data acquisition and data send – to be stored

- `FunctionalTriggerList`

  Partial implementation of a trigger list allowing only items of type `AocsUnitFunctional` to be stored. Some AOCS systems will not have housekeeping interactions with their external units. Such systems should use a `FunctionalTriggerList` since this type of trigger list uses less memory than the `FullTriggerList`.

11.1 The Telemetry Interface

Trigger list objects are telemetry objects because they inherit the `telemeterable` interface. The data they send to the telemetry stream in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	none
Long	instance ID of all object lists they maintain
Debug	same as long TM

11.2 The Reset and Configurable Interface

Trigger list objects inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

The `Resettable` and `Configurable` methods are implemented by trigger list objects by delegation to the corresponding methods of the object lists they maintain

.

12 UNIT TRIGGER OBJECTS

A unit trigger is an active object. Its function is to perform operations on units that need to be performed cyclically. The purpose is to relieve consumers and producers of unit data from the burden of managing bus transactions and buffer refresh operations.

Not all AOCS applications will use unit triggers. Some applications will choose to leave control of data transfers to and from units to the producers and consumers of unit data. Thus, for instance, an attitude controller may be given responsibility for triggering the acquisition of attitude measurements from the sensors and, after polling the sensor to wait for the acquisition to be finished, for performing a refresh operation on the newly acquired data.

If the trigger mechanism were used, then the attitude controller could rely on the converted attitude data being already present in the data pool since the acquisition and refreshing of the data would be done by separate and autonomous trigger objects.

A unit trigger object holds a reference to a trigger list. A trigger list can be seen as a list of operations to be performed on unit objects. Operations can be performed in two phases. When the trigger object is activated, it goes through the units in the trigger lists and performs one or both phases of the operation associated to each of them.

Which operation phase and whether only one or both are performed depends on the trigger object type. Several types of trigger objects can be defined: *normal trigger*, *polling trigger*, *transaction trigger* and *refresh triggers*. Their respective characteristics are:

- *Normal Triggers*

  Normal triggers perform a full data transfer on the unit including both bus transaction and buffer refresh but they assume that the bus transaction operations are of the blocking kind, ie. that the transaction methods return after the bus transaction has been completed.

- *Polling Triggers*

  Polling triggers perform a full data transfer on the unit including both bus transaction and buffer refresh. This means that, on incoming data, they initiate the acquisition transaction, wait for it to be finished and then refresh the acquired data. On the outgoing data, they refresh the outgoing data, initiate the send transaction, and wait for it to be finished.

  The wait for the completion of a transaction is done by polling the `isTransactionFinished` service offered by the unit objects. A time out mechanism is used to decide when to stop waiting for successful completion of a transaction and declare an error that is reported as a failure event.

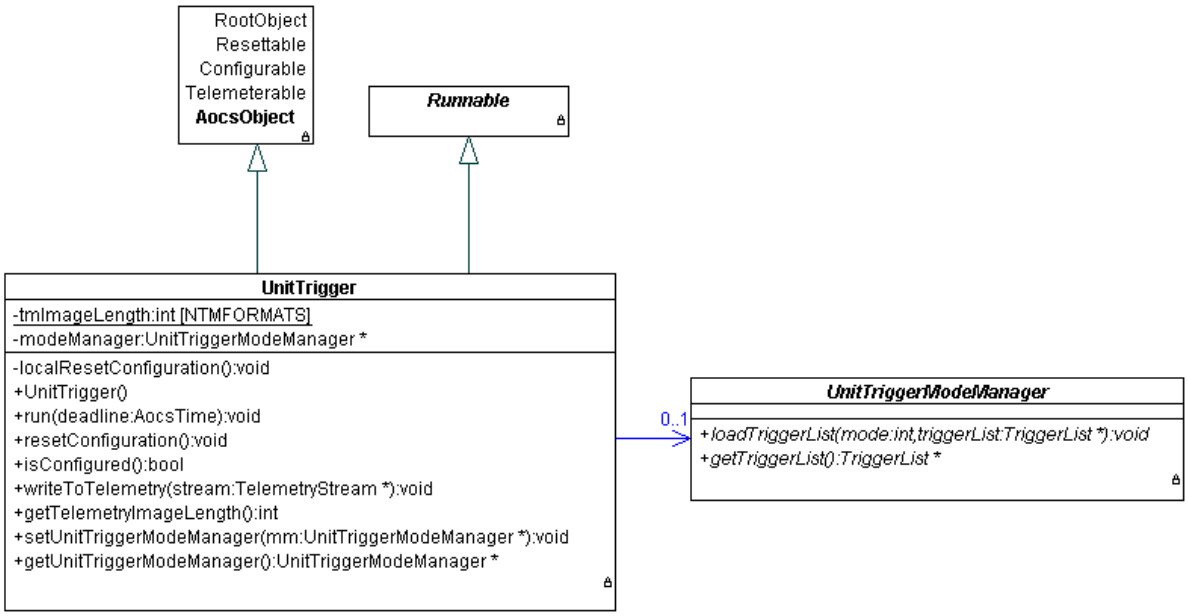
- *Transaction Triggers*

  Transaction triggers perform the bus transaction part of the operation associated to the units in the trigger list. This means that they initiate the bus transactions implied by the items in the trigger list but they do not wait for their completion.

- *Refresh Triggers*

  Refresh triggers perform the buffer refresh part of the operation associated to the units in the trigger list. This means that they call the refresh methods on the unit objects but do not initiate any bus transaction.

The only type of unit trigger supplied by the AOCS prototype framework is the normal trigger encapsulated in class `UnitTrigger`:



Method `run` goes through all the items in a trigger list and performs on each a bus transaction and a refresh operation. The trigger list is provided by a mode manager that implements interface `UnitTriggerModeManager`.

The AOCS prototype framework provides a default unit trigger mode manager that implements the follower mode manager mechanism.

The mode manager performs initialization and shutdown operations on trigger lists that are switched in or out as a result of a mode transition. In an initialization operation units are

powered on (their `switchOn` method is called) and initialized (their `initialize` method is called). In a shutdown operation, units are power off (their `switchOff` method is called).

12.1 The Telemetry Interface

Unit trigger objects are telemetry objects because they inherit the `telemeterable` interface. The data they send to the telemetry stream in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	instance ID of current trigger object
Long	Normal TM + instance ID of unit trigger mode manager
Debug	same as long TM

12.2 The Reset and Configurable Interface

Unit trigger objects inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

Unit triggers have no internal state and therefore do not provide any class-specific Reset method.

Unit triggers provide a class-specific `ResetConfiguration` method that unloads the mode manager component and a class-specific `isConfigured` that returns true if a mode manager component has been loaded.

13 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD6.

13.1 Unit Trigger Mode Manager Plug-In

Name: Unit Trigger Mode Manager **Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `UnitTrigger` class (method `setUnitTriggerModeManager`)
Pre-defined Options: `FollowerUnitTriggerModeManager` component exported by this framelet.
Related Hot-Spots: none
Description Unit triggers need a mode manager to provide them with a trigger list. This hot-spot allows the mode manager to be loaded.

13.2 AOCS Unit Hot-Spot

Name: AOCS Unit Hot-Spot
Visibility Level: framework -level
Adaptation Time: compile-time
Adaptation Method: concrete implementation of class `AocsUnit`
Pre-defined Options: concrete AOCS unit objects exported by this framelet (see section 9)
Related Hot-Spots: none
Description Concrete AOCS units are represented in the AOCS software by proxy objects instantatiated from subclasses of `AocsUnit`. Class `AocsUnit` offers default implementation of all of its methods.

Which ones of these have to be overridden by unit subclasses depends on the unit characteristics. In general, the methods catering to the transfer of data between the hardware buffers and the source and destination buffers should remain unchanged.

13.3 AOCS Hardware Unit Hot-Spot

Name: AOCS Hardware Unit Hot-Spot
Visibility Level: framework –level
Adaptation Time: compile-time
Adaptation Method: implementation of class `AocsUnitHardware`
Pre-defined Options: `MacsTcController` components exported by this framelet implementing an interface to a MACS telecom controller
Related Hot-Spots: none
Description AOCS hardware unit objects encapsulate an interface with the hardware device controlling the communication between the AOCS computer and the external unit. Their implementation is highly device-specific.

13.4 Fictitious Unit Hot-Spot

Name: Fictitious Unit Hot-Spot
Visibility Level: framework –level
Adaptation Time: compile-time
Adaptation Method: implementation of interface `AocsUnitFunctional`
Pre-defined Options: `TorquingThrusters` component exported by this framelet and reconfiguration manager components exported by the unit reconfiguration framelet
Related Hot-Spots: none
Description

Fictitious units are characterized by the `AocsUnitFunctional` interface. Their implementations are highly component-specific.

13.5 Trigger List Hot-Spot

Name: Trigger List Hot-Spot
Visibility Level: framework –level
Adaptation Time: run-time
Adaptation Method: plug-in component in unit trigger mode manager (method `loadTriggerList` in interface `UnitTriggerModeManager`)
Pre-defined Options: none
Related Hot-Spots: none
Description The unit trigger mode manager provides the trigger list to be processed by the unit trigger objects. This hot spot allows the trigger list objects to be loaded onto the unit trigger mode manager.

NB: remove getAddress from AocsUnitHardware

add unit-identifier parameter to getAddress in AocsUnitHardware

trivial implementation for isInitialized in AocsClass: return true

change ángle´ to rate in parameter of GyrProtoype constructor