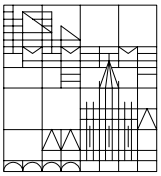


University of Constance
Department of Computer Science

TELEMETRY 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 telemetry framelet. This framelet proposes a solution to the problem of handling telemetry. It defines a telemetry manager component that is completely mission independent and it defines an interface to be supported by all objects that can potentially be sent to telemetry.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.2

Reference: SWE/99/AOCS/003

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 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, To appear in Dec. 2001

RD4 A. Pasetti (2000), *Operational Mode Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/009

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 telemetry framelet for the AOCS framework. The framelet is described at both the framelet concept level and at the framelet architectural level.

This framelet proposes an architectural solution to the problem of managing telemetry data flows in the AOCS software. It defines a re-usable component to act as a telemetry manager and an interface to be supported by all objects that can potentially be sent to telemetry.

The framelet enhances reusability because it decouples the task of managing the telemetry from the layout and format of the telemetry data.

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 RD1 and in particular with the overview of telemetry management.

RD2 described two options for telemetry management.

The baseline option was based on the implementation of `telemeterable` interface by all objects that can potentially be sent to telemetry. Implementation of the interface forces objects to support a method, `writeToTelemetry`, that defines how the objet's state is to be printed to the telemetry stream.

The alternative option regarded printing to telemetry as a form of serialization.

Here the baseline option is retained which is the only one to be presented in this document.

In comparing the present document with RD2, readers should 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 and is only partially applicable to the Java 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. Some specific points to note are:

– The DMA telemetry stream default component (see section 6.4) and the memory section default component (see section 9) are not provided by the Java framework because of lack of support in Java for direct access to memory.

3.3 Notation

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

The class diagrams use UML notation generated with the reverse engineering tool of the *Together* tool (version 4.0).

4 FRAMELET CONSTRUCTS

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

TELEMETRY MANAGEMENT FRAMELET
Abstract Interfaces and Abstract Base Classes
`TelemetryStream` : abstract base class for telemetry streams `Telemeterable` : interface for objects that can write their own state to telemetry `TelemetryModeManager` : interface for the operational mode manager for the failure detection manager.
Core Components
`TelemetryManager` : component encapsulating a telemetry manager (including mode management)
Default Components
`DmaTelemetryStream` : implementation of `TelemetryStream` interface representing a DMA-based telemetry stream `CyclingTelemetryModeManager` : default mode manager for the telemetry manager component implementing a cycling mode management mechanism. `MemorySection` : component encapsulating a range of contiguous memory addresses that are to be copied to the telemetry stream. `TestTelemetryStream` : component simulating a telemetry stream (the telemetry data are sent to a data file).
Design Patterns
Telemetry Management Pattern : design pattern to make an object a telemeterable object

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, interface `Telemeterable` should be implemented by all AOCS

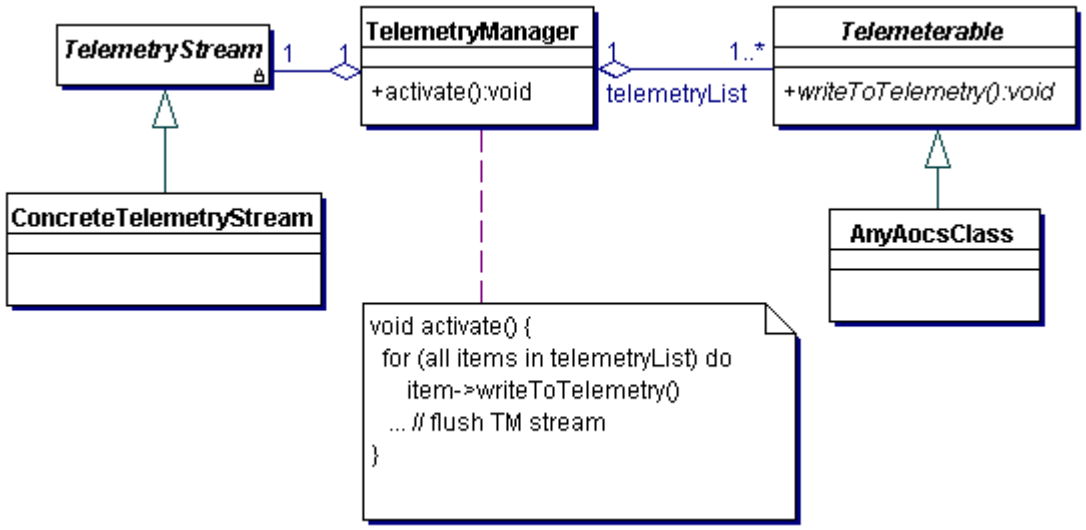
objects. In the prototype framework, however, implementations are only provided for the following objects:

- AOCS data (instances of class `AocsData` and its subclasses)
- AOCS events (instance of class `AocsEvent` and its subclasses)
- Event repositories (instances of class `EventRepository` its subclasses)

5 THE TELEMETRY MANAGEMENT DESIGN PATTERN

This design pattern is introduced to address the problem of separating the management of telemetry data from the layout and content of the telemetry frames. It is based on the manager meta-pattern of RD2.

The pattern is illustrated in the following class diagram:



The telemetry manager maintains a list of references to objects of type `Telemeterable`. Telemeterable objects are objects that are capable of writing their own internal state to the telemetry stream.

The telemetry manager additionally maintains a reference to the telemetry stream. The characteristics of concrete telemetry streams vary widely across AOCS applications and therefore no generic telemetry stream component can be provided. The AOCS framework characterizes telemetry stream through the abstract interface `TelemetryStream`.

5.1 Instantiation of Telemetry Management Pattern

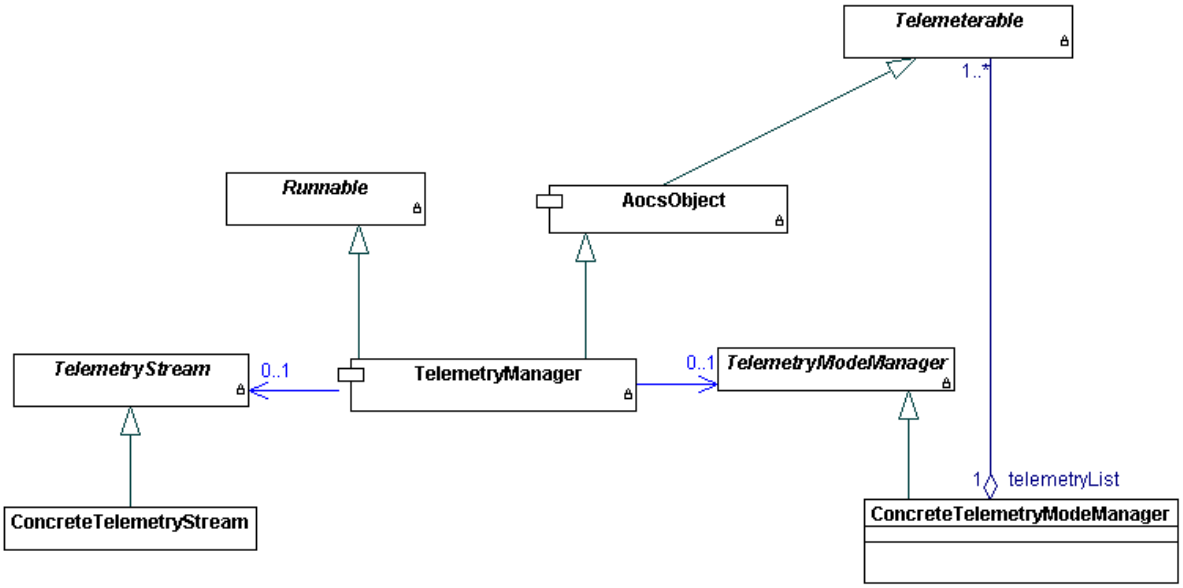
The telemetry management pattern is instantiated in the AOCS framework as follows:

- The telemetry manager is implemented as an active object and its `activate` method is the `run` method it inherits from interface `Runnable`.
- Methods are added to the `Telemeterable` interface to control the format of the telemetry image generated by telemeterable components: it is assumed that telemeterable objects can generate four different telemetry images with different content and layout.

Interface `Telemeterable` allows the telemetry manager (or other components) to select the format of the telemetry image of each telemeterable object.

- The telemetry stream is passed as a reference to the `writeToTelemetry` method. Forwarding of telemetry data to the telemetry stream is thus done directly by telemeterable objects.
- In order to ensure that all non-trivial AOCS objects can potentially have their state included in telemetry, interface `Telemeterable` is implemented by class `AocsObject`.
- The format and type of telemetry data may vary depending on operational conditions. This dependency is modeled by using the operational mode design pattern of RD4 and making the telemetry manager mode-dependent. The list of telemeterable objects then becomes the strategy managed by the telemetry mode manager.

The resulting class diagram is:



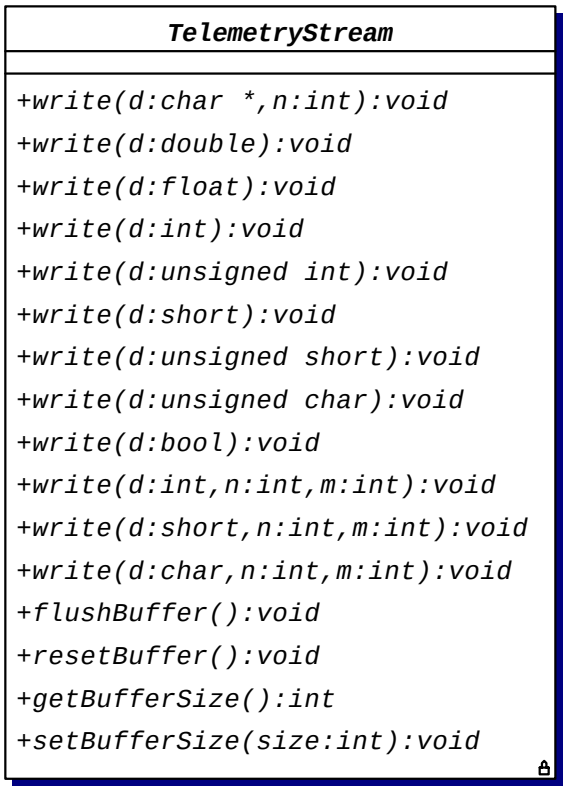
As indicated by the figure, the telemetry manager now gets its list of telemeterable objects from the a telemetry manager whom it sees through the abstract interface `TelemetryModeManager` (see section 8.2).

Interface `Telemeterable` is now implemented by the basic class `AocsObject` which makes all non-trivial objects in an AOCS application telemeterable by default.

6 TELEMETRY STREAMS

The term *telemetry stream* is used to designate the physical entity to which telemetry is written. In most cases, this is a memory buffer where telemetry data are to be copied and from which dedicated hardware will then transfer them in DMA mode to the system bus. In other implementations, the telemetry stream could be an I/O port directly connected to the system bus interface. The telemetry stream interface is independent of its physical implementation but the discussions of implementation details will assume the former, buffer-based, physical model.

From a software point of view, a telemetry stream is represented by an object that implements the `TelemetryStream` interface:



The semantics of the operations defined by this interface are summarized in the following table and further discussed in the following sub-sections:

`write(d)`

Write the value of the argument d to the telemetry stream. Several versions of this method are provided to cater to the following types for d: double, float, int, short, bool
`write(&d, n)`
Write n bytes to the telemetry stream starting address &d. Not implemented in framework prototype.
`write(d,n,m)`
Write a subset of the bits making up the argument d. The bits in the range [n,m] are written. Several versions of this method are provided to cater to the following types for d: int, short, char. Not implemented in framework prototype.
`resetBuffer`
This operation signals to the telemetry stream the beginning of a new telemetry frame. In case of a DMA-based telemetry interface, it causes the telemetry buffers to be reset.
`flushBuffer`
This operation signals to the telemetry stream the end of a frame. Depending on the type of the telemetry interface, after this operation is performed, the telemetry stream may initiate the physical transfer of the data to the system bus.
`getBufferSize, setBufferSize`
Getter and setter methods for the telemetry buffer size. The telemetry buffer size is the maximum number of bytes that can be transferred to the telemetry interface in a single telemetry frame.
`writeFrameNumber`
The telemetry stream keeps track of the frame number. This frame number will generally be included in the telemetry frame (perhaps as header information). This method causes the frame number to be written to the telemetry buffer.
`writeNumberOfBytes`
Telemetry streams handle byte and bit write operations differently. The total number of bytes and of bits in each telemetry frame should normally be written to the telemetry frame itself (in order to permit decoding of the telemetry frame by the ground). This method causes the number of bytes to be written to the telemetry frame.
`writeNumberOfBits`

Telemetry streams handle byte and bit write operations differently. The total number of bytes and of bits in each telemetry frame should normally be written to the telemetry frame itself (in order to permit decoding of the telemetry frame by the ground). This method causes the number of bits to be written to the telemetry frame. Bit writes are not implemented in the framework prototype and therefore this method is also not implemented.

The implementation of the above methods depends on the telemetry interface present in the AOCS framework.

6.1 Telemetry Frames

A batch of data that are treated as a single telemetry packet by the telemetry interface is called a *telemetry frame*.

From the point of view of the AOCS software, a telemetry frame begins when operation `resetBuffer` is performed on the current telemetry stream component and ends when operation `flushBuffer` is performed. All the data sent to the telemetry stream using its `write` operations in between are assumed to be part of the same telemetry frame.

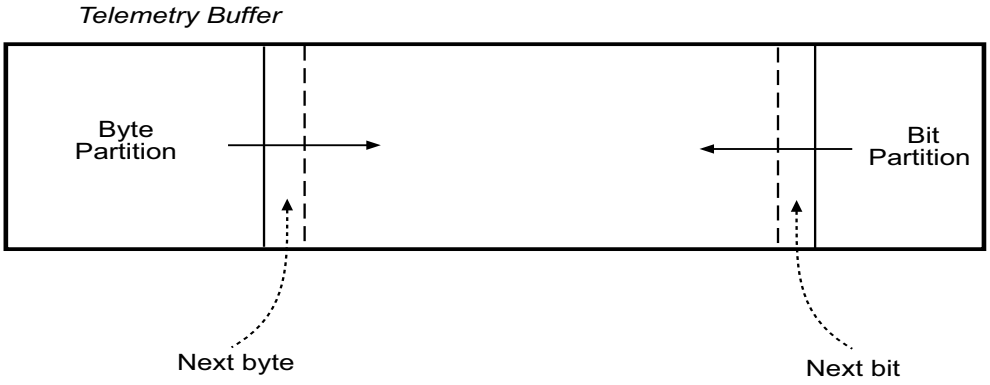
6.2 Error Handling

Telemetry streams are not required to perform any error handling or error checking operations. In particular, they do not check that the amount of data sent to the telemetry stream using its `write` operations is within the capacity of the telemetry channel or of the telemetry buffer. This check should be done by the telemetry manager using the telemetry size parameter that can be retrieved from the telemetry stream.

6.3 Bit and Byte Writes

Telemetry bandwidth is a scarce resource on most satellites. Hence, data should be written to the telemetry stream in as compact a format as possible. For this reason, separate `write` methods are offered that can write either bytes or individual bits. The implementation of the telemetry stream can use this information to optimize the way the data are sent to the telemetry interface. Consider for instance the case of a DMA-based telemetry interface. In that case, the telemetry stream objects puts the telemetry data in a buffer and then the following arrangement can be adopted to use all the available space:

The telemetry buffer is partitioned into two sub-buffers of which one is reserved for byte writes and the second is reserved for bit writes. It is in general not possible to know the size of the bit- and byte-telemetry and it is therefore not possible to assign the partition size *a priori*. One solution is to put byte writes at one end of the buffer and bit writes at the other end as shown in the figure:



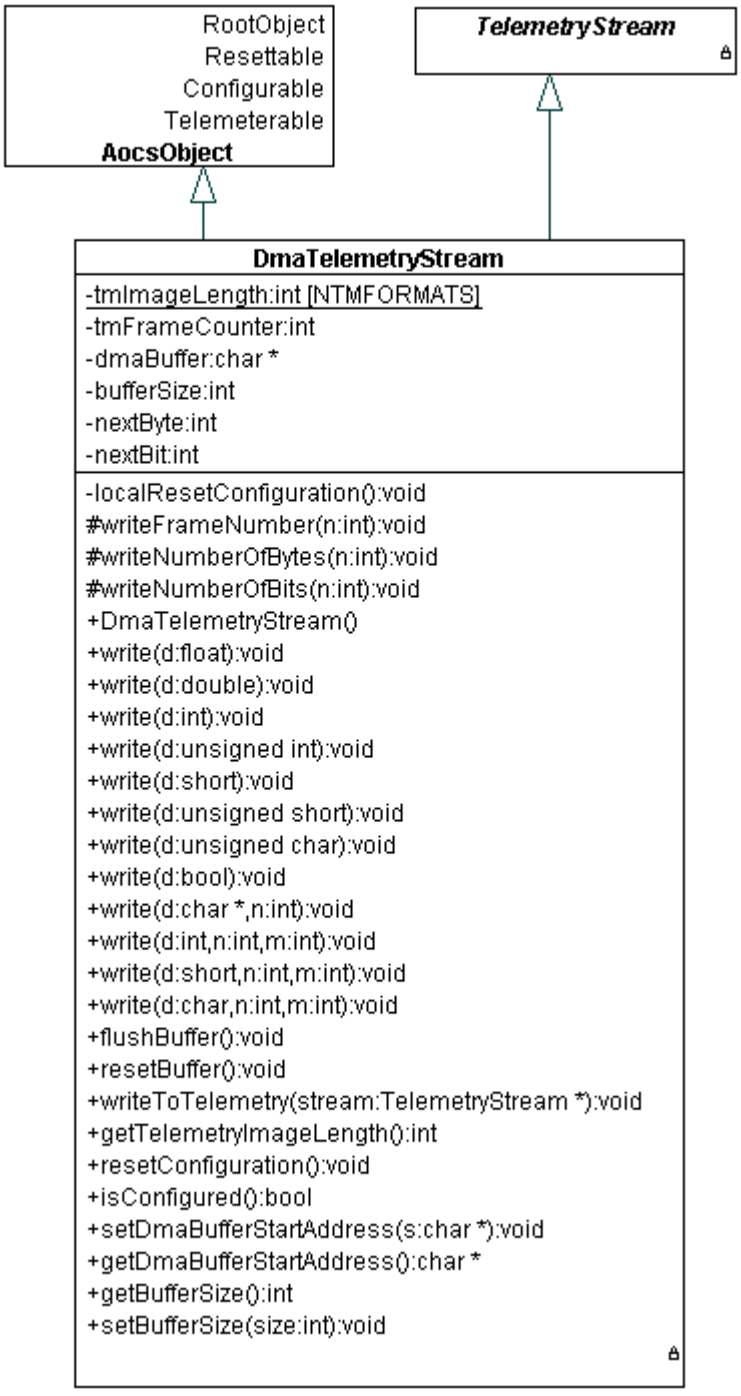
6.4 The DMA Telemetry Stream Component

The framework prototype offers a default implementation of a DMA-based telemetry stream encapsulated in class `DmaTelemetryStream`.

The telemetry interface model that underlies this component assumes that the telemetry data are forwarded to the central on-board computer by a dedicated hardware device that collects them in DMA mode from a pre-defined memory area in the AOCS computer. This pre-defined memory area is called the *DMA buffer*. It is defined by its start address and by its length.

No synchronization mechanism is assumed between the AOCS software and the hardware telemetry interface. When the latter is triggered, it simply collects whatever happens to be in the DMA buffer.

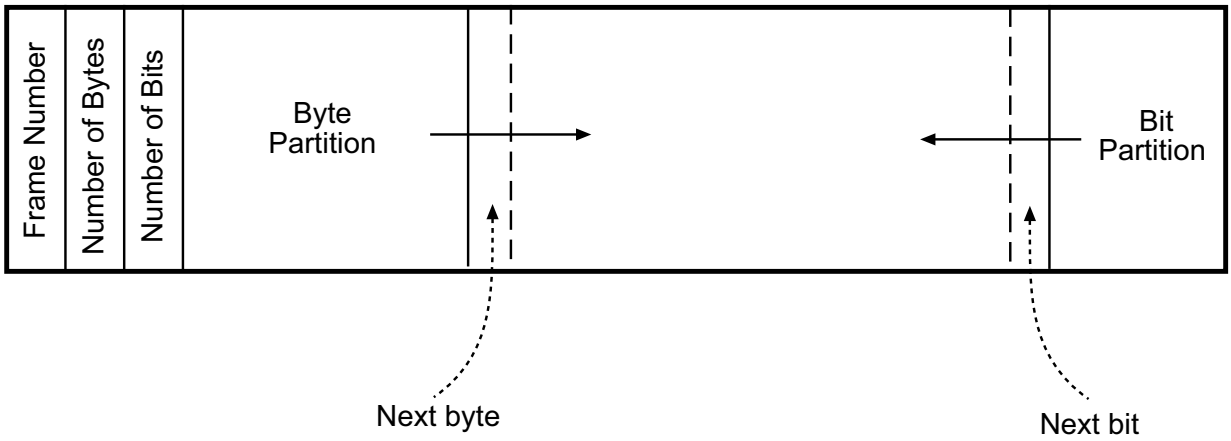
The class diagram for the DMA telemetry stream is:



This class only adds a getter and setter method for the start address of the DMA buffer to the basic operations defined by the `TelemetryStream` interface.

The component assumes a telemetry buffer structure as shown in the figure:

Telemetry Buffer



The telemetry buffer is divided into a byte and bit partition for the reasons explained in the previous sub-section. The first three bytes contain header information, namely:

- byte 1 : frame counter
- byte 2 : number of bytes written in the current frame
- byte 3 : number of bits written in the current frame

Bit operations are however not implemented in the prototype version of this component.

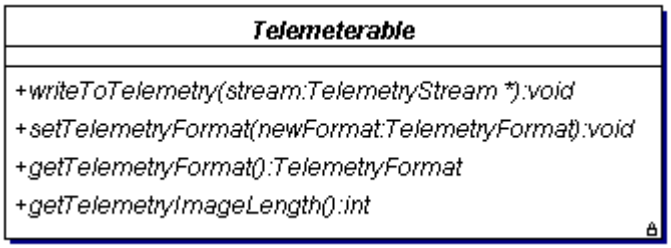
7 TELEMETRY OBJECTS

A *telemetry object* is an object that can potentially be written to telemetry.

Writing an object to telemetry means writing a (subset of) its internal state to the telemetry stream.

7.1 The `Telemeterable` Interface

Telemetry objects implement the `Telemeterable` interface:



The semantics of the operations defined by this interface are summarized in the following table and further discussed in the following sub-sections:

`writeToTelemetry(tmStream)`
A call to this method causes the object to write its own state to the telemetry stream `tmStream`. The method implementation uses the operations defined by the `TelemetryStream` interface to write the object's state to the telemetry stream.
`setTelemetryFormat(newFormat), getTelemetryFormat()`
The type and amount of data written by an object to the telemetry stream when its `writeToTelemetry` method is called depend on the telemetry format of the object. The telemetry format can be changed by calling method `setTelemetryFormat`. Method `getTelemetryFormat` is the corresponding getter method.
`getTelemetryImageLength()`
Return the length in bits of the telemetry image that would be written by the object in response to a call to method `writeToTelemetry`.

The basic method is `writeToTelemetry` that causes the object to write itself to a telemetry stream. This method insulates the telemetry manager from the structure of the telemetry data of each particular object: it confines the information about which data are to be sent to telemetry by each object to the object itself and thus allows the telemetry manager to be written in a generic manner.

Method `getImageLength` returns the length *in bits* of the object's telemetry image. The length is given in bits to take account of both byte and bit `write` operations (see section 6.3).

7.2 Telemetry Formats

The type of information that an object sends to telemetry is defined internally to the object itself by the way it implements the `writeToTelemetry` method. Some control over the quantity of telemetry information is provided by the *telemetry format*.

Telemetry formats are represented by an enumeration type, `TelemetryFormat`. The following formats are foreseen: `short`, `normal`, `long`, and `debugging`. Not all objects implement all formats. Thus the only thing that can be said with certainty about telemetry formats is that: the `short` format does not produce more data than the `normal` format; that the `normal` format does not produce more data than the `long` format; etc.

Note that the telemetry format is a property of each telemetry object, not of the telemetry manager or of the telemetry frame. Hence, the same telemetry frame can include objects with different telemetry formats.

7.3 Telemetry Objects and AOCS Objects

As described in the section on root objects of RD2, the `telemeterable` interface is implemented by `AocsObject`. This means that all but the simplest objects in the AOCS software inherit it and must implement the corresponding methods.

7.4 Telemetry Object Identification

As discussed in greater detail in the next section, the order in which objects are written to telemetry is not fixed: the telemetry manager handles a list of `telemeterable` objects and asks them to write themselves to telemetry in the order in which they are found in the list. The content of the list can – and normally will – be changed dynamically and therefore the content of the telemetry stream cannot be fixed *a priori*.

A way must therefore be provided of identifying the images in the telemetry stream as belonging to a particular object. This is done by stipulating that the first item that an object

writes to the telemetry stream must be its instance identifier followed by the telemetry format for that object. The ground (or the entity interpreting the telemetry stream) can use the instance identifier to determine the class of the object and hence the structure of the data that follow the instance identifier in the telemetry stream.

7.5 Telemetering of Associated Objects

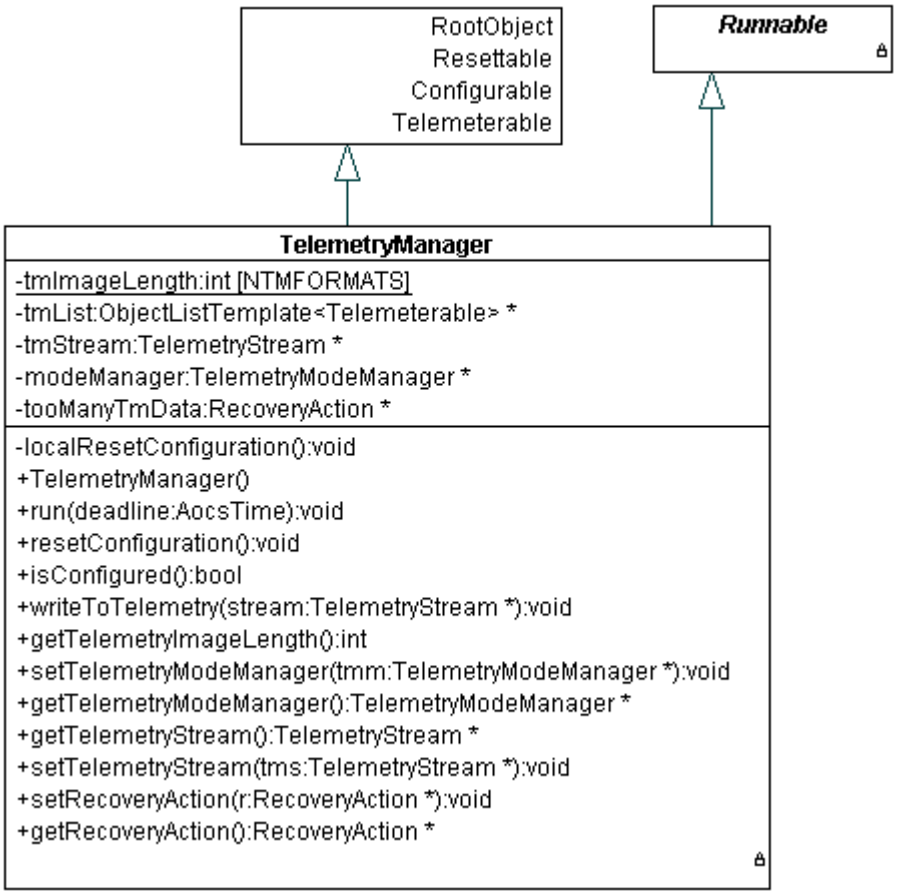
AOCS objects are usually in relationships of associations to each other. If an object A is in a relationship of "strong" association (ie. either aggregation or containment), then its implementation of `writeToTelemetry` will also call the same method on the associated objects.

Thus, for instance, a container object will implement `writeToTelemetry` to call `writeToTelemetry` on all the objects it contains.

8 TELEMETRY MANAGER

The *telemetry manager* is an active component whose responsibility is the management of the objects that must be written to telemetry.

Telemetry managers are instance of class `TelemetryManager`:



The telemetry manager is derived from `AocsObject` and implements interface `Runnable` to signify that it is an active object.

As discussed in section 8.1, the telemetry manager obtains the telemetry list to be used in the next frame from a *telemetry mode manager*.

The public methods specific to this class (ie. not inherited from base classes) are described in the table:

`setTelemetryModeManager, getTelemetryModeManager`
Setter and getter methods for the telemetry mode manager.
`setTelemetryStream, getTelemetryStream`
Setter and getter methods for the telemetry stream where telemetry data are to be written.
`setRecoveryAction, getRecoveryAction`
When it starts to write a telemetry frame to the telemetry stream, the telemetry manager checks that the size of the telemetry frame is compatible with the capacity of the telemetry stream. If this is not the case, a failure event is raised. These are the setter and getter methods for the recovery action associated to this failure.

8.1 Telemetry Manager Implementation

The implementation of the telemetry manager follows the manager pattern of RD2.

The telemetry manager maintains a list of objects that have been marked for inclusion in telemetry. When it is activated, it goes through the list and calls the `writeToTelemetry` method of each object in the list. Additionally, the telemetry manager performs some housekeeping operations of which the most important is to ensure that the size of the telemetry image is compatible with the capacity of the telemetry stream.

A basic implementation for the telemetry manager is:

```
class TelemetryManager : public AocsObject, public Runnable {

  TelemetryList*       tmList;
  TelemetryStream*     tmStream;
  TelemetryModeManager*modeManager;

public:

  void run(AocsTime t){
  {
    // Load the telemetry list to be sent to the TM stream in this frame
    tmList = modeManager->getTelemetryList();

    // Initialize the counter of bytes sent to the TM stream
    int tmDataSize=0;
```

```
    // Reset the telemetry buffer
    tmStream->resetBuffer();

    // Get telemetry buffer size
    int tmStreamSize=8*tmStream->getBufferSize();

    // Send the TM list to the TM stream
    Telemeterable* t;
    for (t=tmList->first(); !tmList->isLast(); t=tmList->next())
    {
         tmDataSize+=t->getTelemetryImageLength();
         if (tmDataSize>tmStreamSize)
         {    . . . // error! raise failure event
              break;
         }
         t->writeToTelemetry(tmStream);
    }

  // Flush the TM buffer
  tmStream->flushBuffer();
}
  // Setter method for the mode manager
  . . .

  // Other methods (reset, etc.)
  . . .
}
```

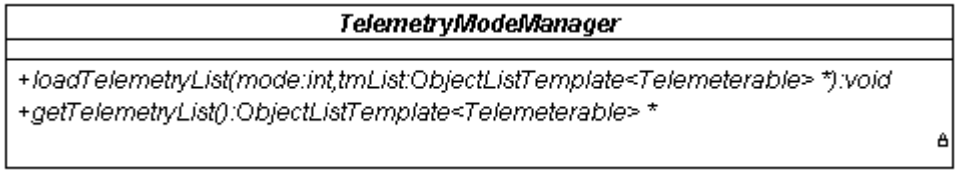
Note that the telemetry list is provided at each activation by a mode manager. The mode manager will typically maintain a set of telemetry lists and will return the one which is appropriate to the current activation. In most cases, the mode manager will simply iterate over a small number of lists representing sub-frames in a telemetry frame.

The mode manager is more likely to be mission-dependent however the prototype framework offers a default telemetry mode manager presented in the next subsection.

8.2 The Telemetry Mode Manager

The telemetry mode manager is a component that is capable of supplying the appropriate telemetry list at any given point in time.

A telemetry mode manager is defined by the following abstract interface:



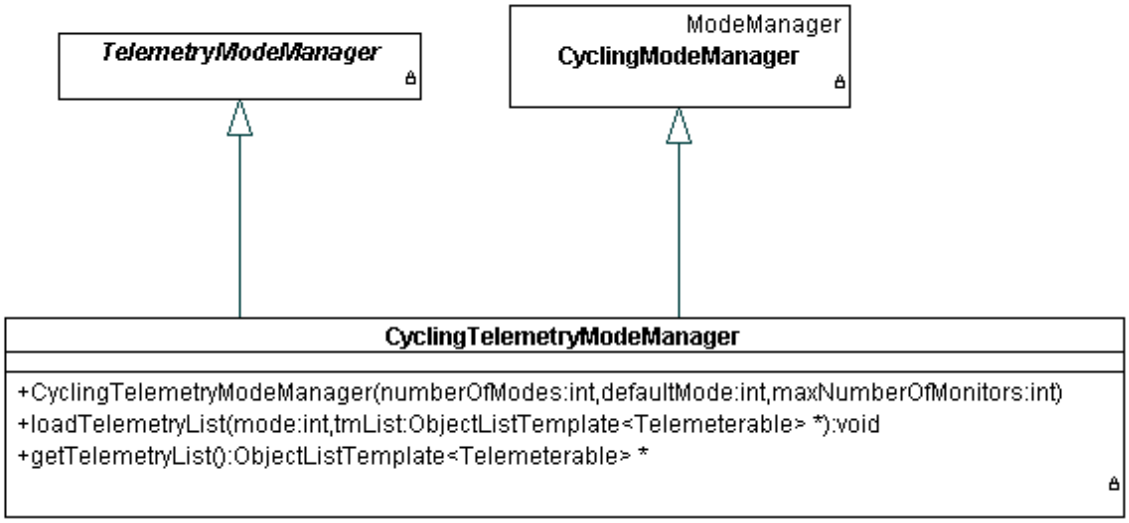
The semantics of the operations defined by this interface are summarized in the following table:

`getTelemetryList()`
This method is called by the telemetry manager to retrieve the currently valid telemetry list.
`loadTelemetryList(int i, ObjectListTemplate<Telemeterable>* t)`
This method is used to configure the telemetry mode manager. It associates telemetry list t to operational mode i.

Concrete telemetry mode managers are defined by the mechanism that they use to decide which particular telemetry list should be returned by method `getTelemetryList` at any given point in time.

The prototype framework provides a default telemetry mode manager that is based on the cycling mode manager. It maintains a set of telemetry lists and cycles through it in a fixed sequence. A call to method `getTelemetryList` returns the currently active telemetry list. Successive calls to this method cause all the telemetry lists loaded in the telemetry mode manager to be returned in a fixed sequence.

The default telemetry mode manager is instantiated from the following class `CyclingTelemetryModeManager`:



Thus, the default telemetry mode manager uses the services offered by the generic cycling mode manager component exported by the operational mode framelet.

8.3 Telemetry Bandwidth Usage

The telemetry budget is often critical and telemetry bandwidth should therefore be used as efficiently as possible. In this respect, it must be stressed that the proposed implementation entails a certain amount of overhead due to the need for each object sent to telemetry to write its instance identifier (see section 7.4) This is the price to be paid for having flexibility in setting the telemetry format. The alternative is to give telemetry lists a fixed content. Flexibility in deciding what is sent to telemetry is then reduced to switching across pre-defined telemetry lists.

A quantitative assessment of the overhead implied by the need to write object's identifiers to telemetry must wait until a full implementation of the AOCS framework.

One way to minimize this overhead is to optimize the implementation of methods `writeToTelemetry`. This is not done in full in the prototype framework.

Another possibility is to compress the data in the telemetry buffer but data compression can probably be done more efficiently at the level of the central satellite computer that gathers the telemetry from all subsystems.

8.4 Telemetry Interface

The telemetry manager is itself a telemetry object because it inherits (indirectly, through `AocsObject`) the `telemeterable` interface.

The data sent to the telemetry stream by a telemetry manager in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	instance ID of current telemeterable list
Long	same as normal TM
Debug	long TM + instance ID of telemetry mode manager and telemetry stream

8.5 The Reset and Configurable Interfaces

The telemetry manager inherits from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

Telemetry managers have no internal state and therefore they do not provided a class-specific implementation of method `reset`.

A call to method `resetConfiguration` unloads the following plug-in components: the telemetry mode manager, the telemetry stream, the recovery action.

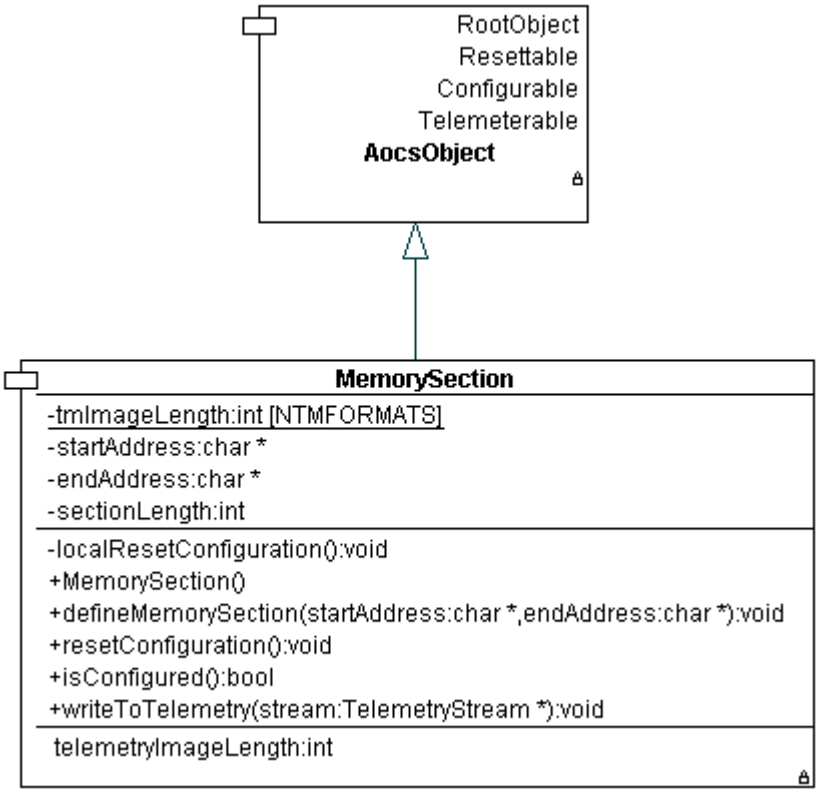
Method `isConfigured` returns true if both the telemetry mode manager and telemetry stream have been loaded.

9 MEMORY SECTION COMPONENTS

The telemetry concept proposed here relies on objects writing their own state to the telemetry stream. However, for maximum flexibility, the option should exist of writing to telemetry a selected section of the AOCS software memory space specified by a starting and end address. For this purpose, the *memory section* components are offered.

A memory section object encapsulates a range of contiguous memory location. Memory sections are `Telemeterable` objects and their implementation of `writeToTelemetry` copies the content of the memory range encapsulated by the memory object to the telemetry stream.

Memory sections are instantiated from class `MemorySection` whose interface is:



The public methods specific to this class (ie. not inherited from base classes) are described in the table:

`defineMemorySection(startAddress, endAddress)`
This method can be used to dynamically define the range of memory locations covered by the memory section.

9.1 Telemetry Interface

Memory sections obviously must be telemetry object since they exist precisely for the purpose of copying a range of memory locations to the telemetry stream.

The data sent to the telemetry stream by a memory section in each telemetry mode are summarized in the table:

TM Format	TM Data
Short	none
Normal	copy of memory locations in the range [`startAddress`, `endAddress`]
Long	same as normal TM
Debug	same as normal TM

9.2 The Reset and Configurable Interfaces

Memory sections inherit from `AocsObject` the `Resettable` and `Configurable` interfaces and must therefore implement the corresponding method.

Memory sections have no internal state and therefore they do not provided a class-specific implementation of method `reset`.

A call to method `resetConfiguration` clears the start and end addresses of the memory range (this equivalent to calling: `defineMemorySection(0,0)`).

Method `isConfigured` returns true if the start and end address of the memory range are not both zero.

10 TEST TELEMETRY STREAM

Telemetry data are written to telemetry streams. A telemetry stream is an object that implements the `TelemetryStream` interface. In concrete applications, telemetry stream objects must interact with the telemetry interface hardware of the AOCS computer. For testing purposes, the AOCS framework offers a test component that implements the `TelemetryStream` interface but sends the telemetry data to a text file.

The test telemetry stream component can be instantiated from class `TestTelemetryStream`. Its output file is called `TestTelemetryStream.txt` and it is created in the same directory in which the test program is run.

For testing purposes, the test telemetry stream can be plugged into the telemetry manager thus causing all telemetry data to be written to the test output file.

11 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD3.

11.1 Telemetry Mode Manager Plug-In

Name: Telemetry Mode Manager **Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `TelemetryManager` class (method `setTelemetryModeManager`)
Pre-defined Options: `CyclingTelemetryModeManager` component exported by this framelet.
Related Hot-Spots: none
Description Telemetry managers need a mode manager to supply them with the list of telemetry objects to be sent to the telemetry stream. This hot-spot allows the telemetry mode manager to be loaded in the telemetry manager.

11.2 Telemetry Stream Plug-In

Name: Telemetry Stream **Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in comp. in `TelemetryManager` class (method `setTelemetryStream`)
Pre-defined Options: `DmaTlemetryStream` component exported by this framelet.
Related Hot-Spots: none
Description Telemetry managers send telemetry data to a telemetry stream. The telemetry stream is

represented by a `TelemetryStream` object that can be dynamically loaded to the telemetry manager. This hot-spot allows the telemetry stream to be loaded..

11.3 Recovery Action Plug-In

Name: Recovery Action **Plug-In**

Visibility Level: framework-level

Adaptation Time: run-time

Adaptation Method: plug-in component in `TelemetryManager` class (method `setRecoveryAction`)

Pre-defined Options: no recovery action is defined by default

Related Hot-Spots: none

Description

When the telemetry manager finds that the current telemetry list has a size too large to fit in the telemetry stream, it raises a failure event. This hot-spot allows the recovery action associated to this failure event to be loaded.

11.4 Telemeterable Hot-Spot

Name: Telemeterable Hot-Spot

Visibility Level: framelet-level

Adaptation Time: compile-time

Adaptation Method: virtual method in `Telemeterable` interface

Pre-defined Options: a default implementation of `Telemeterable` interface is provided by class AocsObject.

Related Hot-Spots: none

Description

The implementation of interface `Telemeterable` defines the type and format of data that a class of objects can send to the telemetry stream. See section 7 for more information.

11.5 Telemeterable List Plug-In

Name: Telemeterable List Plug-In
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `TelemetryModeManager` class (method `setTelemeterableList`).
Pre-defined Options: none
Related Hot-Spots: none
Description The telemetry mode manager maintains a list of telemeterable objects. This hot-spot defines the point where a new list is loaded into a failure detection mode manager.

12 FRAMELET FUNCTIONALITIES

This section defines the functionalities offered by the framelets together with their mutual relationships and their mappings to framelet architectural constructs. The definition follows the guidelines of RD3.

12.1 Conventions

The functionality code defines the type of the functionality according to the following convention:

- CF = can-functionality
- DF = do-functionality
- OF = offer-functionality

The following numbering conventions are used:

- if Fx is a functionality, then the functionalities that are obtained by expanding it are numbered as Fx.n where n is 1, 2, 3, etc
- if CFx is a can-functionality, then the offer-functionality that implement it are numbered OFx,n where n is 1, 2, 3, etc

12.2 Functionality List

The functionalities for the manoeuvre management framelet are shown in the table below. Each entry covers one functionality giving its definition, its relationships to other functionalities (if any) and its mappings to framelets architectural constructs (if any).

DF1	Any component can be made to send a subset of its internal state to the telemetry stream.
	is-implemented-by *Telemeterable design pattern*
DF2	The telemetry management framelet provides a generic and customizable telemetry manager component.
	expands-to *DF2.1 to DF2.5*

DF2.1	The telemetry manager is an active component.
	is-implemented-by *`run()` method in TelemetryManager component*
DF2.2	The telemetry manager provides reset and configuration services.
	is-implemented-by *TelemetryManager component* ***uses*** *CF1 from the system management framelet (reset services)* ***uses*** *CF2 from the system management framelet (configuration services)*
DF2.3	The telemetry manager is a telemeterable component
	is-implemented-by *TelemetryManager component* ***uses*** *CF3 from this framelet*
DF2.4	The telemetry manager maintains a set of telemetry lists. A telemetry list is a list of components whose state must be included in telemetry. At any given time, only one telemetry list is active. When it is activated, the telemetry manager directs the components in the currently active telemetry list to send their state to the telemetry stream.
	is-implemented-by *TelemetryManager component*
DF2.5	The telemetry manager checks that the size of the telemetry image associated to the current telemetry list is compatible with the capacity of the currently selected telemetry stream. If this is not the case, then a failure event is generated.
	is-implemented-by *TelemetryManager component* ***uses*** *DF1 from inter-component communication framelet (error reporting mechanism)*
CF3	Any component can be made telemeterable, namely it can become a component whose state can be sent to telemetry.
	matches *the Telemeterable hot spot*

CF4	The telemetry manager can be customized to send any combination of component states to the telemetry stream.
	expands-to *CF4.1 and CF4.2*
CF4.1	The telemetry manager can be customized to manage any number of telemetry lists.
	matches *the Telemeterable List Hot-Spot*
CF4.2	The telemetry manager can be customized to use any algorithm to select the active telemetry
	matches *the Telemetry Mode Manager Plug-in Hot-Spot* ***uses*** *DF1 from the operational mode management framelet (mode management design pattern)* ***is-implemented-by*** *OF4.1*
OF4.1	The telemetry management framelet offers a telemetry mode manager component implementing a cycling mode management mechanism to cycle through a fixed number of telemetry lists.
	matches *the Telemetry Mode Manager Plug-In Hot-Spot* ***uses*** *OF1 from the operational mode management framelet (the CyclingModeManager)* ***is-implemented-by*** *the CyclingTelemetryModeManager component*
CF4	The telemetry manager component can be customized to send the telemetry data to any telemetry stream.
	matches *the Telemetry Stream Plug-in Hot-Spot* ***is-implemented-by*** *OF4.1*
OF4.1	The telemetry management framelet offers a component encapsulating a DMA-based telemetry stream. This telemetry stream assumes that telemetry data are collected by a DMA mechanism from a pre-defined buffer.

	matches *the Telemetry Stream Plug-in Hot-Spot* ***is-implemented-by*** *the DmaTelemetryStream component*
CF5	The telemetry manager can be customized to associate any recovery action to the failure check of DF2.5.
	matches *the Recovery Action Plug-In Hot Spot* ***uses*** *DF2 from the inter-component communication framelet (association of recovery actions to failure events)*
OF6	The telemetry management framelet offers a component to allow a range of contiguous memory locations to be written to the telemetry stream.
	matches *the Telemeterable Hot-Spot* ***is-implemented-by*** *the MemorySection component*