AOCS FRAMEWORK - TEST REPORT

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 was tested by using it to generate the software for a prototype AOCS. This document defines the test environment that was used to test the AOCS prototype and the results of the tests.

Written By:	A. Pasetti
Date:	30 April 2002
Issue:	1.1
Reference:	SWE/00/AOCS/001

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 Deleted

RD4 A. Pasetti (2000), *Methodological Issues*, AOCS Framework Document ref. SWE/99/AOCS/018

RD5 A. Pasetti (2000), *Inter-Component Communication Framelet* AOCS Framework Document ref. SWE/99/AOCS/005

RD6 A. Pasetti (2000), *Object Monitoring Framelet*, AOCS Framework Document ref. SWE/99/AOCS/008

RD7 A. Pasetti (2000), *Data Processing Framelet*, AOCS Framework Document ref. SWE/99/AOCS/006

RD8 A. Pasetti (2000), *AOCS Unit Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/017

RD9 A. Pasetti (2000), *Reconfiguration Management Framelet* AOCS Framework Document ref. SWE/99/AOCS/015

RD10 A. Pasetti (2000), *Operational Mode Management Framelet* AOCS Framework Document ref. SWE/99/AOCS/009

RD11 T. Brown, A. Pasetti (2000), *Manoeuvre Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/012

RD12 A. Pasetti (2000), *Failure Detection Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/010

RD13 A. Pasetti (2000), *System Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/021

RD14 A. Pasetti (2000), *Failure Recovery Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/011

RD15 A. Pasetti (2000), *Telemetry Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/003

RD16 A. Pasetti (2000), *Telecommand Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/014

RD17 A. Pasetti, T. Brown (2000), *Controller Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/016

RD18 A. Pasetti (2000), *AOCS Prototype Definition*, AOCS Framework Document ref. SWE/99/AOCS/020

RD19 *MACS Bus Handbook*

RD20 A. Pasetti (2000), *AOCS Prototype Definition*, AOCS Framework Document ref. SWE/99/AOCS/020

RD21 A. Pasetti (2000), *Framework Instantiation*, AOCS Framework Document ref. SWE/00/AOCS/002

RD22 ERC32 Home Page, *http://www.estec.esa.nl/wsmwww/erc32/freesoft.html*

RD23 RTEMS Home Page, *http://www.rtems.com/RTEMS/rtems.html*

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
MIMO	Multi-Input-Multi-Output
NM	Normal Mode
NTT	Non-Time-Tagged
OBDH	On-Board Data Handling system (aka as OBDS)
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
SISO	Single-Input-Sinle-Output
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 test environment that was used to test the *AOCS prototype* instantiated from the *AOCS prototype framework*. The AOCS prototype is a simplified AOCS application which is implemented using the constructs offered by the AOCS prototype framework. The AOCS prototype thus serves as a test bed for the AOCS prototype framework.

3.1 Context

The context for the definition of the prototype framework is the architectural design of the full framework. This is presented at system concept definition level in RD2 and at framelet concept and at framelet architectural definition level in RD5 to RD17. The prototype framework is defined in RD18. The AOCS prototype whose testing is described here is presented in RD20.

3.2 General Test Approach

The AOCS prototype was tested first in a test environment developed and run under Visual Studio and then on the ERC32 simulator. These test environments will be referred to as the *VS Test Environment* and the *ERC32 Test Environment*.

The main constraint on the testing was the absence of a real-world simulator simulating the satellite dynamics and the AOCS sensors and actuators. Testing was therefore done in open loop: the test harness stimulates the sensors and sends telecommands to the AOCS and intercepts the commands that the AOCS sends to the actuators and collects the telemetry.

A second constraint was the failure to simulate the hardware interfaces of the AOCS computer (the MACS, telemetry and telecommand interfaces).

A third constraint was the lack of an interface to visualize the state of the AOCS under test and to issue commands to the AOCS while the test is under way.

The second and third constraints were addressed by introducing an extra task in the AOCS software called the *Test Harness*. The test harness runs as the last task in the AOCS software. Its job is to perform the following actions:

– to prepare and load the stimuli for the MACS units for the next AOCS cycle

– to prepare and load the telecommands for the next AOCS cycle

– to collect the data sent to the MACS units in the previous cycle and store them in a dedicated buffer

– to collect the telemetry data generated by the AOCS software in the last cycle and store them in a dedicated buffer

At the end of a test, the MACS and telemetry data collected by the test harness are sent to a file in the VS test environment and to the standard output device in the ERC32 test environment (the ERC32 simulator has no file system but can send data to the console).

3.3 Test Types

Three types of tests were performed on the prototype AOCS:

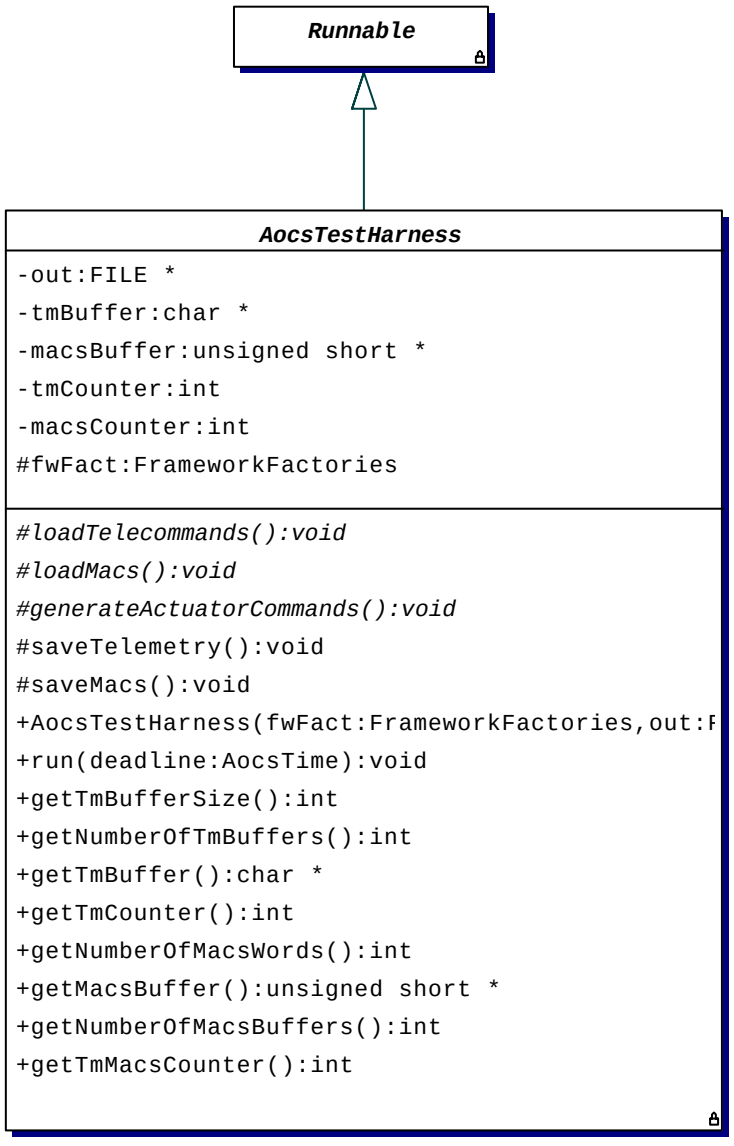
– *Timing Tests* with the aim of measuring the overhead introduced by the framework and the execution time of the prototype AOCS

– *Memory Tests* with the aim of measuring the memory overhead introduced by the framework and the memory requirements of the prototype AOCS

– *Functional Tests* with the aim of demonstrating that the functionalities offered by the AOCS framework are correctly implemented.

4 THE TEST HARNESS

A test harness is encapsulated as an object ultimately derived from the base class `AocsTestHarness`:



Class `AocsTestHarness` implements interface `Runnable` to allow its instances to be treated as active objects that can be scheduled alongside other AOCS tasks. Its method `run` (the entry point for the scheduler) is structured as follows:

```
void AocsTestHarness::run(AocsTime t)
{
     loadTelecommands();
     loadMacs();
     generateActuatorCommands();
     saveTelemetry();
     saveMacs();
}
```

Method `loadTelecommands` assembles the telecommands to be sent to the AOCS under test and places them as telecommands packets in the telecommand buffer from where they are picked up by the telecommand loader. This method therefore simulates the telecommand interrupt servicing routine.

Method `loadMacs` assembles the stimuli for the MACS units representing the sensor read-outs for the next cycle. The stimuli are assembled as 16-bit words with the same format with which the data would be received from the MACS bus. They are loaded into the hardware registers of the MACS units using method `setFcHwInpBuff` exposed by class `AocsUnit`. Method `loadMacs` therefore simulates the MACS bus interrupt servicing routines that would normally be responsible for recovering data received from the MACS bus.

Method `generateActuatorCommands` generates the commands for the actuators and loads them in the attitude data pool in the locations corresponding to the spacecraft torque request. Thus, this method overrides the torque requests generated internally to the AOCS software by its attitude controllers.

Method `saveTelemetry` collects the telemetry generated by the AOCS under test in the last cycle and stores it in a dedicated buffer (`tmBuffer`) where it remains available for inspection at the end of the test.

Method `saveMacs` collects the outgoing MACS data generated by the AOCS under test in the last cycle and stores them in a dedicated buffer (`macsBuffer`) where it remains available for inspection at the end of the test. The MACS data are retrieved by accessing the outgoing hardware buffers of the MACS unit objects in the AOCS software using method `getFcHwOutBuff` exposed by class `AocsUnit`.

The base class `AocsTestHarness` provides implementations for methods `saveTelemetry` and `saveMacs`. For the other three methods only trivial default implementations are provided. Meaningful implementations are to be provided by subclasses.

The implementation of methods `loadTelecommands` and `loadMacs` define a test profile because they define the stimulation of the AOCS software as a function of time. A subclass of

`AocsTestHarness` therefore defines a test case by providing implementations for these two methods.

4.1 Default Test Cases

Two default test cases are provided as subclasses `AocsTestHarness_1` and `AocsTestHarness_2`. The first one was only used for initial confidence testing and is not documented further. The test profile of `AocsTestHarness_2` is instead describe below.

The physical spacecraft configuration simulated by test case `AocsTestHarness_2` is:

- Nominal attitude with the z-axis sun pointing and x- and y-axis perpendicular to the sun vector
- Constant angular rate `w_x` applied to the x-axis and no angular rate around either of the other two axes.

The spacecraft physical configuration data are used to compute the inputs for the AOCS sensors. For all sensors but FSS_B, these inputs are computed assuming perfect calibration. For FSS_B, a scaling error of 10% is instead assumed.

The `AocsTestHarness_2` lasts 10 AOCS cycles (10 seconds).

The test case `AocsTestHarness_2` simulates the uplink of the following telecommands to the AOCS:

TC Name	Cycle	Telecommand Description
TC1	3	Load an attitude slew manoeuvre with the following characteristics: slew around x-axis with angular rate of `-w_x` starting immediately and extending to the end of the test
TTC2	4	Transaction telecommand consisting of simple telecommands TC2.1 and TC2.2
TC2.1		Change the operational mode of the AOCS Mission Mode Manager to FSP
TC2.2		Reconfigure the FSS
TC3	6	Reconfigure the FSS

The second column gives the AOCS cycle when the telecommands are uplinked. All the telecommands carry a zero time tag which means that they are executed immediately.

Finally, this test harness simulates a gyro failure at the second cycle when the gyro A output is changed from 0 to a value greater than the maximum permitted physical gyro output.

4.2 Close Loop Simulation

As explained in section 3.2, no closed loop tests were performed because no real-world simulator for the AOCS was available. However, should such a real-world simulator become available in the future, it could be easily integrated in an `AocsTestHarness` subclass. Its implementation of `loadMacs` would then compute the stimuli for the MACS bus as a function of the latest batch of MACS command generated by the AOCS software and collected by subroutine `saveMacs`.

5 THE VS TEST ENVIRONMENT

The VS test environment is used to verify the AOCS prototype from a functional point of view only. Tests in this environment are intended as preparation for the tests in the ERC32 environment.

The main test program is in file `AocsPrototype.cpp`. This main program performs five actions:

- It opens an output file where the test results will be sent
- It instantiates the framework
- It instantiates a test harness
- It runs the instantiated AOCS application
- It saves the MACS and telemetry data collected during the test run in a file

The instantiation process is described in RD21. The following three steps are described in the following subsections.

5.1 Test Harness Instantiation

The test harness instantiation is performed as follows:

```
AocsTestHarness* aocsTestHarness =
                        new AocsTestHarness_2(fwFact, out);
```

Note that the test harness is treated throughout the test program as a pointer to the super class `TestHarness`. Hence changing test harness only requires changing this single line of code.

5.2 Running the AOCS Application

The framework instantiation results in nine `Runnable` components being created:

- the telemetry manager
- the telecommand manager
- the telecommand loader
- the failure detection manager
- the failure recovery manager
- the controller manager
- the manoeuvre manager

- the acquisition unit trigger
- the send unit trigger

Additionally, the test harness must also be treated as a Runnable object as explained in section 3.2.

All the Runnable objects are loaded in array:

```
Runnable* aocsTasks[N_AOCS_TASKS];
```

A cyclical scheduler is then simulated by the following code:

```
for (int i=0; i<N_AOCS_CYCLES; i++)
     for (int j=0; j<N_AOCS_TASKS; j++)
          if (aocsTasks[j]!=NULL)
               aocsTasks[j]->run(0);
```

The inner loop represents an AOCS cycle the outer loop controls the number of AOCS cycles that are simulated by the test program.

5.3 MACS and Telemetry Data Save

At the end of each AOCS cycle, the test harness buffers up the MACS and telemetry data generated during the cycle. Up to MACS_SAVE and TM_SAVE cycles of MACS and telemetry data can be thus buffered.

Before exiting, the test program calls functions `flushMacsBuffers` and `flushTelemetryBuffers` which copy the buffered MACS and telemetry data to an output file. Only minimal formatting is done on the MACS and telemetry data.

Two types of telemetry data are printed to the output file: event repository telemetry images and the attitude data pool telemetry image.

The print out of an event repository telemetry image has the following format:

Event repository name
Event repository instance identifier
Event repository telemetry format
Event repository counter
Event repository entries in sequence starting from the most recent one

The print out of the attitude data pool has the following format:

Data pool name
Data pool identifier
Data pool format
Data pool entries in the order in which they were loaded into the data pool. For the attitude data pool this is: FssX FssY Gyr Sas1x Sas1y Sas2x Sas2y Sas3x Sas3y RwSpd1 RwSpd2 RwSpd3 RwSpd4 TorqueX TorqueY TorqueZ RwAngMomX RwAngMomY RwAngMomZ RwTor1 RwTor2 RwTor3 RwTor4 ThuOn1 -> ThuOn6 ThuDel1-> ThuDel6 AttEst1 AttEst2 AttEst3

The MACS words are printed in hex format as they would have appeared on the MACS bus. They represent the wheel torques and the thruster commands (the delay and on times) as they were sent to the SAP unit. The order in which they are printed is as follows:

Rw1Torque -> Rw4Torque, SapA1DelTime -> SapA6DelTime, SapA1OnTime -> SapA6OnTime, SapB1DelTime - SapB6 DelTime, SapB1OnTime -> SapB6OnTime

6 THE ERC32 TEST ENVIRONMENT

The ERC32 test environment is based on the ERC32 simulator (see RD22) integrated with the RTEMS operating system (see RD23).

The ERC32 version of the AOCS prototype application is instantiated and controlled by code in file `AocsPrototypeTestErc32.cpp`. Except for this file, all the other application files are the same as for the VS version of the AOCS prototype. Where necessary, and only very rarely, compiler switches are used to make small and localized changes in the code within these files.

The ERC32 AOCS prototype is organized as follows:

- One initialization task (entry point: `Init`) instantiates the framework and the test harness
- One quasi-cyclical task (entry point: `AocsTask`) runs all the `Runnable` components in the application (including the test harness) and then suspends itself for 1 second.

Both the initialization and the quasi-cyclical task entry points are in file `AocsPrototypeTestErc32.cpp.`

This file additionally configures the RTEMS operating systems through `#define` control switches. These are taken from the standard RTEMS examples with the two following changes:

- The initialization task stack size is set equal to 2*RTEMS_MINIMUM_STACK_SIZE or 5 kBytes. A value of 2.5 kBytes was also tried but was found to be insufficient.
- Usage of the floating point is enabled with the RTEMS_FLOATING_POINT control switch.

Finally, since `AocsPrototypeTestErc32.cpp`

is a C++ file and RTEMS expects to link into C modules, C-style linkage is specified for the initialization and quasi-cyclical task entry points.

6.1 The Initialization Task

The initialization task in `AocsPrototypeTestErc32.cpp` performs the following actions:

- It instantiates the framework
- It instantiates a test harness
- It loads an array of `Runnable` components

- It creates and starts the quasi-cyclical task

The framework instantiation is done as described in RD21.

The test harness instantiation is done as for the VS environment (see section 5.1).

The array of `Runnable` components is called `aocsTasks`[]. Its purpose is to hold the pointers to all the `Runnable` components in the AOCS application. It is used by the quasi-cyclical task to activate the Runnable components in sequence as discussed in the next sub-section.

The quasi-cyclical task creation is done using RTEMS services. As in the case of the initialization task, the stack size is set at 2*RTEMS_MINIMUM_STACK_SIZE and the floating point co-processor option is specified.

6.2 The Quasi-Cyclical Task

In each activation, this task goes through the list of items in the `aocsTasks` array and calls their run method in sequence. In this sense, it simulates a non-preemptive scheduler.

After N_AOCS_CYCLE activations, the procedures to flush the MACS and telemetry buffers are called. This is the same operation as described in section 5.3.

This task is said to be quasi-cyclical because at the end of each cycle, it suspends itself for 1 second and the duration of the suspension does not take its execution time into consideration. A truly cyclical task would need to use the `rtems_task_wake_when` service[1] or it should arrange for a timer to kick it at the desired frequency.

[1] But note that, according to the RTEMS user's guide, the granularity of the rtems_task_wake_when service is only 1 second.

7 FUNCTIONAL TESTS

The functional test have the aim of demonstrating that the functionalities offered by the AOCS framework are correctly implemented. A functional test is encapsulated by one test harness object (see section 4). Only one functional test was performed corresponding to test harness TestHarness_2 described in section 4.1.

The success of the test is verified by performing the checks listed in the next three subsections. The list gives the parameters that needs to be checked in the test output file (see section 5.3) and their expected value or behaviour.

In analyzing the test output, care should be given to the fact that there is a one cycle delay in the reporting of data. This is due to the fact that the test harness task (that generates the simulated inputs for the AOCS software and collects its outputs) is the last task to run in each AOCS cycle.

Functional tests were performed both in the VS and the ERC32 environments. Their results in the two environments should be the same.

7.1 Event Repository Checks

1. Since the MACS bus is not simulated, failures should be generated every time the AOCS starts a MACS transaction. This can be verified by checking the event counter for the failure event repository. It should increase by 25 in each AOCS cycle (except when there are mode changes or unit switch over, see below). The event indicator in the failure event buffer should be 146 (NO_BUS_TRANSMISSION)
2. No configuration events should be generated at any time during the test. This can be verified by checking that the event counter of the configuration event repository remains equal to 0.
3. The mode event repository should indicate that the telemetry mode manager changes mode at every AOCS cycle. This can be checked by verifying that its event counter increases by 1 in every cycle (unless other mode changes occur) and that the component undergoing the mode change has an ID of 119.
4. In telemetry frame 7, the mode event repository should indicate that all mode managers have undergone a mode change as a result of telecommand TC2.1 (see section 4.1). Note that there are 6 mode managers in the AOCS prototype.
5. No events should be reported by the system manager.

6. In telemetry frame 5, a gyro reconfiguration should occur as a result of the gyro output error being detected by the failure detection manager and corrected by the failure recovery manager. This can be verified by checking that the reconfiguration event repository indicates a reconfiguration of reconfiguration manager with instance ID equal to 232 (gyro reconfiguration manager).

7. In telemetry frame 5, an FSS reconfiguration should occur as a result of telecommand TC2.2 (see section 4.1). This can be verified by checking that the reconfiguration event repository indicates a reconfiguration of reconfiguration manager with instance ID equal to 194 (FSS reconfiguration manager)

8. No reconfigurations should occur as a result of TC3 (see section 4.1) since the FSS, by the time telecommand, is received has no more healthy configuration available.

9. In telemetry frame 7, four events should be generated in the telecommand event repository corresponding to TC_LOADED and TC_SUCCESS for TC1 and TC2.

10. In telemetry frame 9, two events should be generated in the telecommand event repository corresponding to TC_LOADED and TC_FAILURE for TC3. The TC fails because the FSS had already reconfigured.

11. In telemetry frame 7, a “manoeuvre started” event should be generated in the manoeuvre event repository. The manoeuvre was loaded by TC1 with a delayed time tag.

12. At least one recovery event should be generated for each failure event. Since the number of failure events exceeds the capacity of the failure event repository at every cycle (because of MACS failures), the number of recovery events generated in each cycle should always be equal to the capacity of the failure event repository.

7.2 Attitude Data Pool Checks

13. The FSS_X and all the SAS outputs except SAS_1_Y output in the attitude data pool should nominally be zero. In the first cycle, they should be equal to the unit bias corrections (the test harness inputs only show through from the second cycle) and from the seventh cycle, the FSS_X output should reflect the scaling error in FSS_B (FSS_B is switched in by telecommand TC2.1).

14. The FSS_Y and the SAS_1_Y outputs in the attitude data pool should reflect the simulated rotation of the spacecraft around the x-axis. Up to the sixth telemetry frame, they should be identical. Afterwards, the FSS output should be 10% smaller reflecting the fact that FSS_B has been switched in by telecommand TC2.1 and that it has a 10% scaling error.

15. The gyro output in the attitude data pool should be 0 except in telemetry frame 4 where the effect of the simulated gyro failure is reflected.
16. The torque requests in the attitude data pool should be equal to the torque generated by the attitude controller with the FSS and GYR inputs.
17. The reaction wheel speed in the attitude data pool should be zero.
18. The reaction wheel angular momentum in the attitude data pool should be zero.
19. The reaction wheel torques in the attitude data pool should be zero up to telemetry frame 6 when the AOCS goes into FSP mode where wheels are used as actuator. Afterwards, the torques on wheels RW1 to RW3 should be equal to the torque generated by the FSP attitude controller with the FSS and GYR inputs.
20. In the first six frames, when the AOCS is in CSP mode and the thrusters are used as actuator, the thruster on-times in the attitude data pool should be equal to the torque generated by the CSP attitude controller with the FSS and GYR inputs. From frame 7 onward, the thruster on-time should remain frozen to a constant value.
21. The thruster delay times in the attitude data pool should always be equal to zero.
22. The attitude errors in the data pool should always be equal to zero.
23. The attitude set points should be zero up to telemetry frame 7 after which the x component increases linearly (it is set by the manoeuvre loaded with TC1).

7.3 MACS Buffer Checks

24. The reaction wheel torque requests in the first four MACS buffers (ie. as long as the AOCS is in CSP mode) should be zero. Afterwards, their value should be equal to the torque generated by the FSP attitude controller with the FSS and GYR inputs.
25. The thruster delay times in the MACS buffers should be identically zero.
26. The thruster on-times in the first four MACS buffers (ie. as long as the AOCS is in CSP mode) should reflect the torque generated by the CSP attitude controller with the FSS and GYR inputs. Afterwards they should remain frozen to the value they had in the fourth MACS buffer.

8 TIMING TESTS

Memory test were performed with the aim of measuring the timing overhead introduced by the framework and the timing requirements of the prototype AOCS. These tests were only performed in the ERC32 environment.

Timing tests are performed by running program `TimingTestErc32`. This is a slightly modified version of `AocsPrototypeErc32` (see section 6). It differs from the latter in the following respects:

- The test harness is not run
- In each cycle (ie. in each call of the quasi-cyclical task), the AOCS runnable components are run 100 times and the number of RTEMS processor ticks at the beginning and at the end of the 100 cycles are collected and sent to the output device.

The tick measurements are used as a proxy for the processor execution time. The measurement is done over 100 cycle because of the poor granularity of the tick measure (one tick is equal to 10 ms).

Two timing measurements were made, for the full AOCS prototype and for the functionality managers only, as described in the next two sections.

All timing measurements were made with the ERC32 running at 14 MHz.

The compilation was performed with the debugging option disabled and with the –O3 optimization option. It should also be stressed that when the framework was implemented, no special attention was given to ensuring timing efficiency and hence it is likely that a review of the code might lead to some further optimizations.

8.1 Full AOCS Prototype Timing Measurements

The measurement procedure described above was applied to the fully instantiated AOCS prototype. It was found that one AOCS cycle requires 4.8 ms of processor time.

8.2 Functionality Managers Timing Measurements

In order to evaluate the timing overhead introduced by the framework, the above timing measurement procedure was repeated with an “empty” AOCS application instantiated from the framework as follows. All functionality managers are created but they are left unconfigured. Thus, for instance, the telemetry manager is created but its telemetry lists are not loaded with any telemeterable objects. The empty AOCS applications can be obtained by commenting out the following statements from procedure `instantiateAocs` (see RD21):

```
loadTelemetryLists(fwFact);
loadFailureDetectionLists(fwFact);
loadUnitTriggerLists(fwFact);
loadControllerLists(fwFact);
```

The empty AOCS application is thus an AOCS application where the functionality managers are activated and run normally but do not have any clients upon which to operate. The empty AOCS application represents the smallest and fastest executable application that can be instantiated from the framework.

The timing test indicate that one cycle of the empty AOCS application requires 0.2 ms of processor time. This is the timing overhead introduced by the framework machinery in an AOCS application instantiated from it.

9 MEMORY TESTS

Memory test were performed with the aim of measuring the memory overhead introduced by the framework and the memory requirements of the prototype AOCS. These tests were only performed in the ERC32 environment.

The compilation was performed with the debugging option disabled and with the –O3 optimization option. It should also be stressed that when the framework was implemented, no special attention was given to ensuring memory efficiency and hence it is likely that a review of the code might lead to some further optimizations.

9.1 Basic Memory Requirement Measurements

Memory requirements were measured in the following manner:

- The components of interest are linked into a single executable
- The executable is loaded into the ERC32 simulator (SIS command) and the code and data memory requirements as reported by the SIS tool are recorded.

Different measurements were made corresponding to different sets of components of interest.

The make file for the memory test can be found in directory `AocsFrameworkHome/MemoryTest`. The make file can build three executables:

- `EmptyMemoryTest`: this executable is built from a dummy main program. Its memory requirements essentially correspond to the memory requirements of the C++ run-time systems.
- `MemoryTestFullFw`: this executable links together all the components provided by the framework including both the core and default components.
- `MemoryTestOnlyFunctMan`: this executable links together the framework functionality managers and no other plug-in components. It therefore represents the core that has to be included by any AOCS generated from the framework.

The memory requirements of these modules as reported by SIS are shown in the table below.

	`EmptyMemoryTest`	`MemoryFullFw`	`MemoryTestOnlyFunctMan`
`.text`	30 Kbytes	165 Kbytes	74 Kbytes
`.data`	2 Kbytes	64 Kbytes	19 Kbytes

`.bss`	0	5 Kbytes	2 Kbytes

From the table, the following considerations can be evinced:

- The framework functionality managers, after subtraction of the "empty memory test" requiremens, represent in a sense the "overhead" introduced by the framework and require a total of: (74-30)+(19-2)+2=63 Kbytes
- The `MemoryFullFw` module links together a total of 112 components. Thus the "average" memory requirement of a framework component is: ((165-30)+(64-2)+5) / 122=1.7 Kbytes

9.2 AOCS Prototype Memory Requirement Measurements

The memory requirements of the AOCS prototype were measured as above by loading the AOCS prototype executable in the SIS environment.

The memory requirements reported by the SIS environment for the `AocsPrototypeTestErc32` module are:

```
.text      260 Kbytes
.data      73 Kbytes
.bss       25 Kbytes
```

Note that the above figures include:

- the C++ run-time system
- the RTEMS system
- the instantiation code (object factories)
- the framework code proper.

The AOCS prototype tests were run with a stack of 5 Kbytes. A stack of 2.5 Kbytes was also tried but was found to be insufficient. Note, however, that all objects were created on the heap during the initialization phase. Thus stack usage can be expected to be low.

9.3 Heap Usage Measurement

The heap usage of the AOCS prototype application was measured by placing the following code sections at the beginning and end of the instantiation code:

```
int* a = new int;
int startOfHeap = (int)a;
printString(out,"Start of Heap: ");
printInt(out,startOfHeap);
```

```
printString(out,"\n");

. . .    // framework instantiation code

int* b = new int;
int endOfHeap = (int)b;
printString(out,"End of Heap: ");
printInt(out,endOfHeap);
printString(out,"\n");
```

The difference between the value of: `startOfHeap` and the value of: `endOfHeap` gives the total heap usage of the AOCS application (recall that memory is only allocated during the framework instantiation phase). This was found to be equal to: 59Kbytes.