**XMM OPTICAL MONITOR**

MULLARD SPACE SCIENCE LABORATORY
UNIVERSITY COLLEGE LONDON

H.E.Huckle, P.J.Smith, M.C.R.Whillock

# XMM-OM ICU FM SOFTWARE DETAILED DESIGN

Document: XMM-OM/MSSL/SP/0205.03

**Distribution:**

| | | |
|---|---|---|
| XMM-OM Project Office | A Dibbens | Orig |
| ESA PX | H Eggel | ✓ |
| CSL | P Rochus | |
| | S Roose | |
| Los Alamos National Laboratory | C.Ho | ✓ |
| UCSB | T.Sasseen | ✓ |
| Royal Greenwich Observatory | R Bingham | |
| University College London | J Fordham | |
| Mullard Space Science Laboratory | R Card | |
| | M Carter | ✓ |
| | R Chaudery | |
| | R Hunt | |
| | H Huckle | ✓ |
| | H Kawakami | |
| | T Kennedy | ✓ |
| | D Self | |
| | P Smith | ✓ |
| | P Thomas | |
| | M Whillock | ✓ |
| | K Mason | |
| | A Smith | |

Author: Date:

OM Project Office Date:

Distributed: Date:

## CHANGE RECORD

| Issue | Date | Comments |
|---|---|---|
| 1 | 30 Oct 1998 | First Edition |
| 2 | 5 Nov 1999 | Added Mode Manager to overview diagrams.<br>Corrected connectivity in overview diagrams.<br>Corrected Summary of main s/w components table.<br>Added Timer A interrupt handler to component summary.<br>Added descriptions of<br>a) task priority structure<br>b) interrupt structure<br>c) exception handling and debugging<br>d) use of pragmas.<br>Added section on Timer Delay correction.<br>Added section on use of adaref1750a.<br>Corrected bootstrap variable locations<br>Corrected bootstrap routine locations<br>Changed definition of prime/redundant<br>Added bootstrap variables location sentence.<br>Removed references to SAFING package, now part of TMQ.<br>Additional comments added to all operational code module descriptions.<br>Additional comments added to some basic mode code module descriptions. |
| 3 | 12 May 2000 | Update for release 10 including:<br>a) Automatic Safing in the event of F/W Position Loss<br>b) Automatic selection of focus heaters as a function of filter<br>c) Prevention of transition from Safe mode without f/w in blocked position<br>d) Prevention of HV ramp up without f/w in blocked position. |

# TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1 Purpose

This document specifies the detailed design of the software contained within the Flight Model (FM) of the Instrument Control Unit (ICU) of the Optical Monitor (OM) instrument onboard the ESA spacecraft XMM (X-ray, Multi-Mirror) mission.

It's purpose is to provide an understanding of the basic design of the software, and show that it is capable of meeting the requirements set out in the Software Requirements Document RD XMM-OM/MSSL/SP/0024.01.

The intended readership is includes :-

1. The technical development team for this software, in order to aid clarification of the software structure and show top level compliance with the requirements.
2. Other OM team members, including PI, project manager, system engineers, software management, PA, test managers, EGSE & operations personnel, COI's, and others to whom requirements, schedule, interfaces, and quality are relevant.
3. ESA, as they will assume responsibility for operating and supporting the software from about 6 months after launch up to the end of the mission (perhaps 10 years).
4. Anyone else who is interested, including other XMM experimenters & users.

## 1.2 Scope

The scope of this document is limited to a detailed description of the ICU onboard software associated with the OM instrument. The ICU is primarily concerned with providing overall system control, spacecraft interface data handling, and instrument monitoring.

It does not include OM onboard DPU software. The DPU software is primarily responsible for the scientific data collection, processing and forwarding to the ICU.

## 1.3 Definitions, Acronyms and Abbreviations

| | |
|---|---|
| CCD | Charge Coupled Device (detector) |
| CONFIG | CONFIGuration |
| DBI | Digital Bus Interface (between OM & spacecraft) |
| DBU | Digital Bus Unit |
| DDD | Detailed Design Document |
| DEM | Digital Electronics Module |
| DMA | Direct Memory Access |
| DPU | Digital Processing Unit |
| DSP | Digital Signal Processor |
| EGSE | Electrical Ground Support Equipment |
| EOB | Electro-Optical Breadboard (development phase) |
| EPIC | European Photon Imaging Camera |
| ESA | European Space Agency |
| FIFO | First-In First-Out (queue) |
| FOV | Field Of View |
| HK | Housekeeping (data/information) |
| ICB | Instrument Control Bus |
| ICU | Instrument Control Unit |
| I/O | Input-Output |
| MACSbus | Modular Attitude Control System bus |
| NHK | Non-periodic Housekeeping |
| OBDH | On-Board Data Handling (system) |
| OM | Optical Monitor (instrument) |
| PSU | Power Supply Unit |
| RAM | Random Access Memory |
| RBI | Remote Bus Interface (from OM to spacecraft) |
| RGS | Reflection Grating Spectrometer |
| ROM | Read Only Memory |
| S/C | Spacecraft |
| S/W | Software |
| SSI | Serial Synchronous Interface |
| TBA | To Be Added |
| TBC | To Be Confirmed |
| TBD | To Be Defined |
| TC | Telecommand queue |
| TM | Telescope Module |
| TM | TeleMetry queue |
| TMPSU | Telescope Module Power Supply Unit |
| UV | Ultra-Violet |
| XMM | X-ray Multi-Mirror Instrument |

## 1.4 References

| | |
|---|---|
| Ref - 1. MSSL XMM-OM User Requirements Specification, | XMM-OM/MSSL/SP/0030.01 |
| Ref - 2. MSSL XMM-OM On-Board Software Requirements, | XMM-OM/MSSL/SP/0024.01 |
| Ref - 3. ESA XMM EID Part-A, | RS-PX-0016 |
| Ref - 4. ESA XMM EID Part-B, | RS-PX-0018 |
| Ref - 5. ESA XMM EID Part-C, | RS-PX-0024 |
| Ref - 6. OBDH System | RS-PX-0015 |
| Ref - 7. Packet Structure Definition | RS-PX-0032 |
| Ref - 8. XMM-OM ICU S/W Architectural Design | XMM-OM/MSSL/SP/0059 |
| Ref - 9. XMM-OM ADA Coding Standard | XMM-OM/MSSL/SP/0008 |
| Ref - 10. XMM-OM ICU EGSE and S/W Development. Environment | XMM-OM/MSSL/SP/0025 |
| Ref - 11. XMM-OM User Manual (EM) | XMM-OM/MSSL/SP/0005 |

# 2. ICU Software

## *2.1 Overview*

The overall instrument function is provided by the instrument controller. Its main software functions are as follows:-

- Configuring the instrument.
- Monitoring for breakdown/failure conditions (and safing if required).
- Controlling and monitoring status of, the detector, the telescope power supply and the DPU
- Incorporating new or modified code modules for itself or the DPU
- Collecting and telemetering. instrument housekeeping and engineering packets.
- Accepting, reformatting into packets and telemetering science data from the DPU
- Interfacing with the OBDH for data and commands.
- Monitoring and controlling the thermal environment.

The ICU software consists of 3 programs :-

| | |
|---|---|
| **BOOTSTRAP** | This resides in ROM and is copied into RAM for execution. It is responsible for bringing up the ICU in a known safe state after turn on or spacecraft initiated reset, from either a cold or warm start. It also copies the basic state software from ROM to RAM. |
| **BASIC** | This resides in ROM and is copied into RAM for execution. Basic will be responsible for loading the uplinked ICU operational mode code into RAM, housekeeping and basic thermal control. |
| **OPERATIONAL** | This is uplinked and will reside in RAM. Operational provides the full functionality of the ICU. It also allows up-linking of the DPU DPUOS code to provide full OM |

## 2.2 Main Software Components for Basic and Operational.

The diagrams overleaf illustrate the control and data flows between the main software components for both basic and operational code. A brief explanation of each component is also given. These two modes share many components. Their similarities and differences are summarised below, together with the type of telecommands (and Task Identifier - TID - if appropriate) they service.

| Component | Type of TC | TID (HEX) | Function in Basic | Function in Operational |
|---|---|---|---|---|
| DEMPSU | 5 | 80 | • Resets DEMPSU Latchup<br>• Turns-on DPU if Off | • Resets DEMPSU Latchup<br>• Turns-on DPU if Off |
| DETECTOR | 5 | 10 ⇒ 18 | ABSENT | • Control and monitor detector. |
| DPU CONTROLLER | 5 | A4 ⇒ A6 | ABSENT | • Uses SSI to communicate with the DPU.<br>• Configure and control DPU modes.<br>• Control Science and Engineering data flow from DPU and send to TM QUEUE.<br>• Monitors DPU heartbeats<br>• Turns off DPU |
| HK | 5 | D0 | • Collect and pass HK packets to the TM QUEUE that monitor only the TMPSU and DPU heartbeats. | • Collect and pass HK packets to the TM QUEUE for the whole OM. |
| ICB | 5 | 41 | • Controls dataflow to/from the instrument subsystems using the ICB interface | • Controls dataflow to/from the instrument subsystems using the ICB interface. |
| MECHANISMS | 5 | 60, 65 | ABSENT | • Control & monitor mechanisms (filter wheels, dichroic). |
| MEMORY MANAGER | 6 | - | • Supports memory uplink and downlink and memory checksum calculations for the ICU only | • Supports memory uplink and downlink for the DPU only. |
| MODE MANAGER | 5 | - | • Implements mode change request to Safe | • Implements mode change requests from spacecraft |
| RBI | 5,10 | 50 | • Provides routines to support the RBI chip<br>• Handle appropriate interrupts to the TC and TM queues and time.<br>• Supply Watchdog Facility | • Provides routines to support the RBI chip<br>• Handle appropriate interrupts to the TC and TM queues and time.<br>• Supply Watchdog Facility |
| SSI | see DPU | - | • Monitors DPU heartbeats and sends the count and DPU status to the HK. | • Passes control and data info to the DPU using the SSI interface.<br>• Obtains info from the DPU using the SSI interface. |

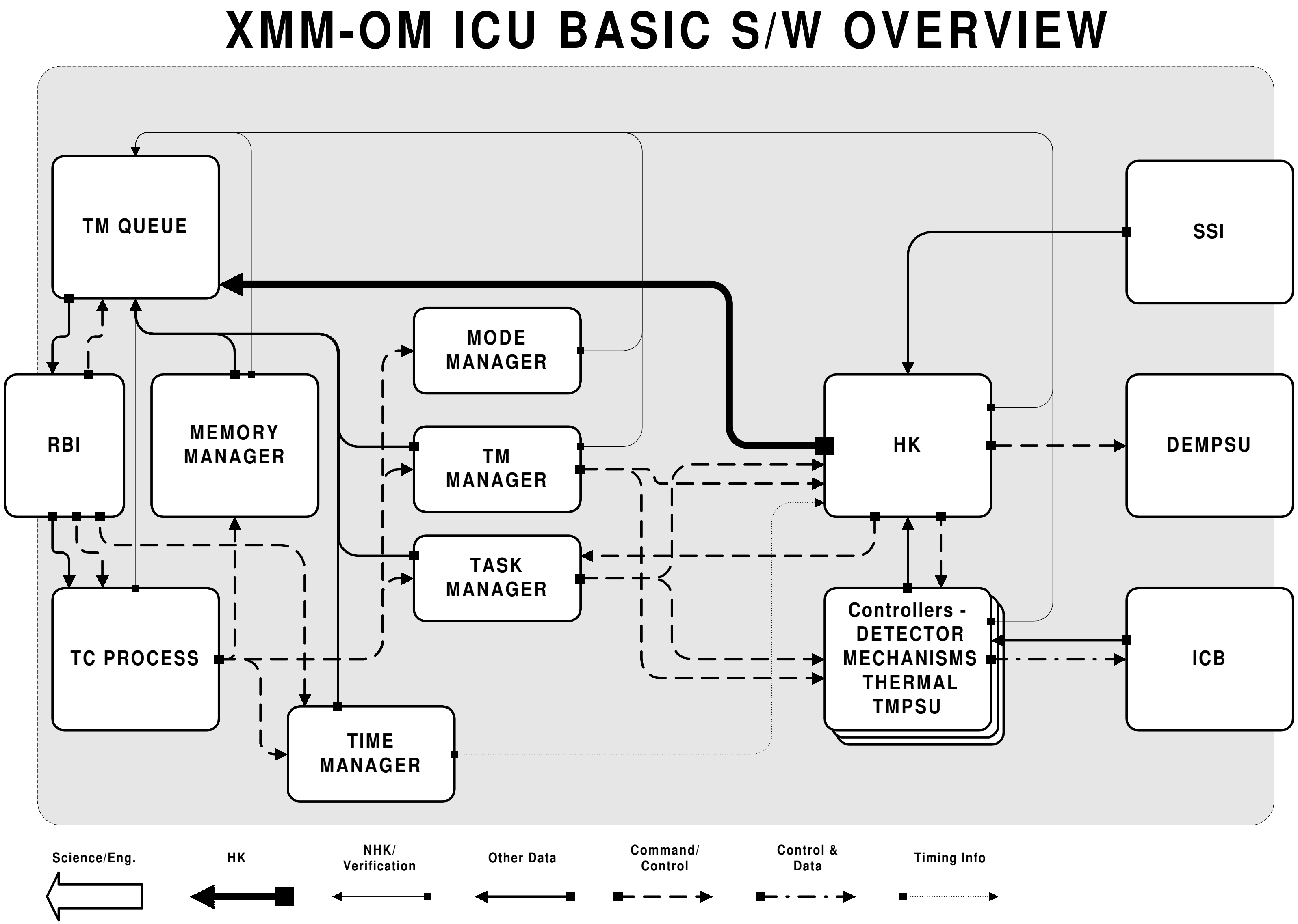
Continued on next page…

| Component | Type of TC | TID (HEX) | Function in Basic | Function in Operational |
|---|---|---|---|---|
| TASK MANAGER | 5 | | • Implements the task management packet requests | • Implements the task management packet requests |
| TC PROCESS | All | | • Obtains telecommand packets from the telecommand queue.<br>• Verifies, acknowledges and routes telecommand packets - the 'main' program | • Obtains telecommand packets from the telecommand queue.<br>• Verifies, acknowledges and routes telecommand packets - the 'main' program |
| THERMAL | 5 | 66,67 | • Enables or disables Main and Forward Heaters simultaneously. | • Provide full thermal control |
| TIME MANAGER | 10 | - | • Implements the Time management packet requests (verification and synchronisation).<br>• Provide time stamps for packets. | • Implements the Time management packet requests (verification and synchronisation).<br>• Provide time stamps for packets. |
| TEMEMETRY MANAGER | 9 | - | • Enables/Disables packets defined by their SID'S | • Enables/Disables packets defined by their SID'S |
| TM QUEUE | Supplies TM | - | • Provide ability to control and queue telemetry packets for downlink. | • Provide ability to control and queue telemetry packets for downlink.<br>• Initiates Safing of HV if TM queue remains full for > 5 mins |

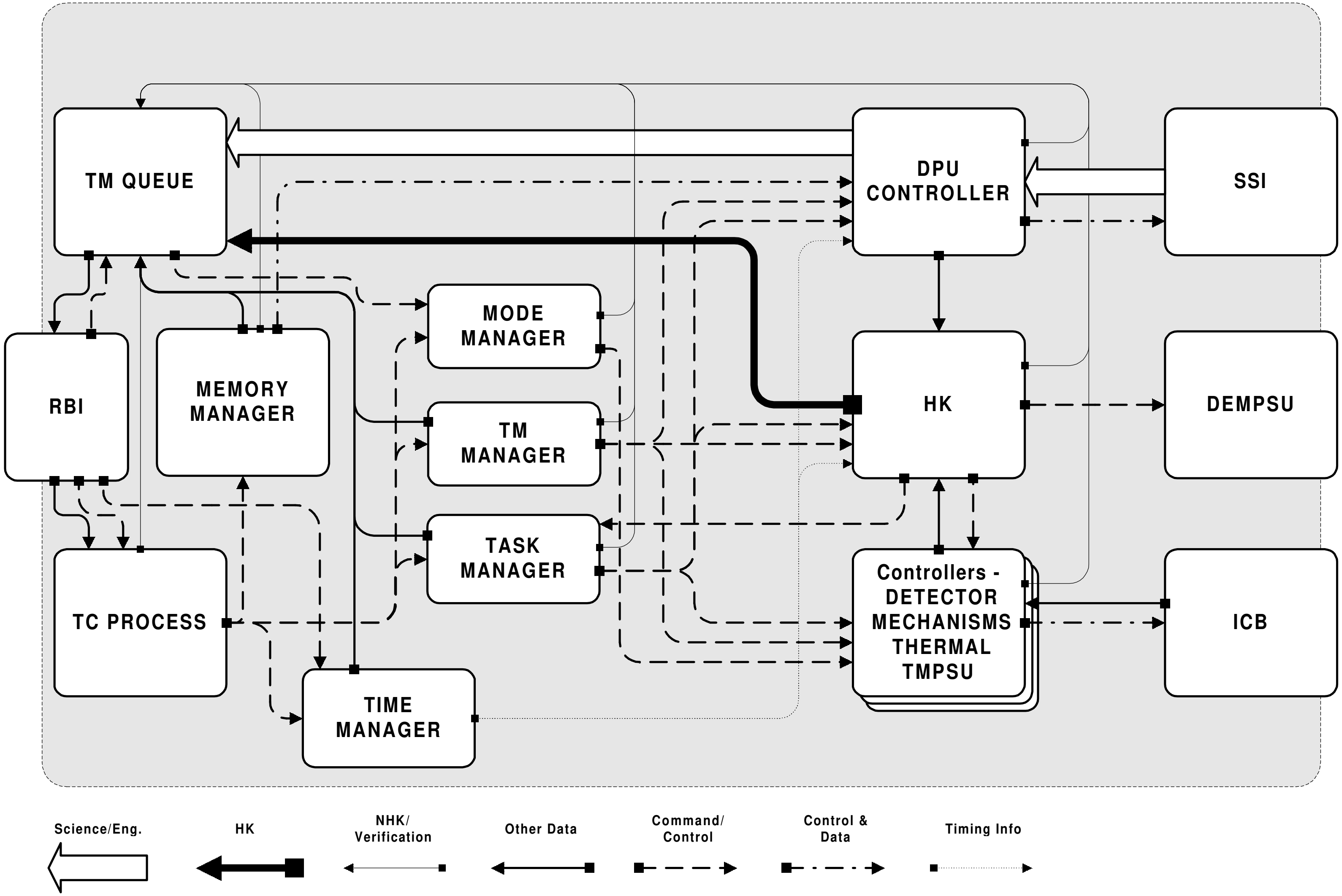
## 2.3 Principle Memory Areas

| Code Address (hex) | | Description |
|---|---|---|
| Start | End | |
| 0 | 3FF | Bootstrap |
| 400 | 3FFF | Basic Mode |
| 3800 | FFFF | Operational Mode |

| Data Address (hex) | | Description |
|---|---|---|
| Start | End | |
| 2C7 | 2D4 | Bootstrap Deduced ICU Status |
| 2F1 | 313 | Bootstrap Filter Wheel Acceleration Table |
| 3F2 | 3FD | Memory Loading Work Area |
| 400 | 403 | RBI Communications Area (CCA) |
| 404 | 5F3 | TC Queue |
| 5F4 | 9FF | TM Queue |
| A00 | A0A | RBI Code Work Area |
| A0B | A0B | SSI Code Work Area |
| FC0 | FCF | DEBUG Area |
| 1000 | 1B55 | Basic Mode Operands |
| 1C00 | 4A10 | Operational Mode Operands |
| 23A4 | 23DB | Focus Heater Settings as function of Filter Wheel |
| E900 | FD00 | Main Program Stack |
| FD01 | FFFF | Interrupt Stack |


XMM-OM ICU BASIC S/W OVERVIEW
TM QUEUE
SSI
MODE MANAGER
RBI
MEMORY MANAGER
TM MANAGER
HK
DEMPSU
TASK MANAGER
Controllers - DETECTOR MECHANISMS THERMAL TMPSU
TC PROCESS
ICB
TIME MANAGER
Science/Eng.
HK
NHK/ Verification
Other Data
Command/ Control
Control & Data
Timing Info

# XMM-OM ICU OPERATIONAL S/W OVERVIEW



## 2.4 Task Priorities

ADA tasks are allocated to actions that are executed in parallel with other actions. In order to ensure that the behaviour of tasks is a deterministic as possible, their priorities (defined in the package IMPORTANCE) are allocated in bands as follows:

| **Task Type** | **Priority Band** |
|---|---|
| H/W Simulators (for debugging) | 191 -> 200 |
| RBI Watchdog reset | 190 |
| S/W Watchdogs | 171 -> 189 |
| "Semaphore" Tasks | 131 -> 140 |
| "Monitor Tasks" (e.g. DPU, TC) | 111 -> 130 |
| "Working Tasks" e.g. HK, Blue | 011 -> 110 |
| "Idle" Tasks (default) | 010 |

The tasks, and the packages that contain them, are as follows:

| **Task** | **Package** | **Basic** | **Operational** |
|---|---|---|---|
| HV_PROCESS[2] | DETANALOG | - | Yes |
| LOAD_CENTROID_TABLE_TASK[3] | DETDIGITAL | - | Yes |
| LOAD_WINDOW_TABLE_TASK_TYPE[4] | DETDIGITAL | - | Yes |
| HEARTBEAT_WATCHDOG[5] | DPU | - | Yes |
| DATA_MANAGER[6] | DPU | - | Yes |
| CONTROL[7] | HEATER | - | Yes |
| PROCESS[8] | HK | Yes | Yes |
| GUARDED[1] | ICB | Yes | - |
| TCPROC[9] | ICU (the main program) | Yes | Yes |
| MEMORY_DUMP[10] | ICU_MEM_MANAGER | Yes | Yes |
| MECH[11] | MECHANISM | - | Yes |
| SAFING_TASK[12] | MODEMAN | - | Yes |
| SEMAPHORE[13] | MUTEX | Yes | Yes |
| WATCHDOG[14] | RBI | Yes | Yes |
| BCP4[15] | TIME_MAN | Yes | Yes |
| GUARDED[1] | TMQ | Yes | - |

Notes.
1. In operational code, the function of this task (to perform controlled access to a resource) is provided by the SEMAPHORE task in package MUTEX..
2. Ramps the HV voltages up and down.
3. Loads the Blue Electronics centroid look-up table.
4. Loads the Blue Electronics window table.
5. Monitors the DPU heartbeats and issues alerts in there absence.
6. Monitors and processes all output from the DPU.
7. Monitors and controls the telescope module heaters.
8. Acquires Housekeeping.
9. Monitors the Telecommand stream.
10. Performs memory load and dumps.
11. Controls the mechanisms.
12. “Safe”s the instrument (HV down, filter wheel to blocked).
13. Provides emulation of a mutex type semaphore.
14. Controls the RBI watchdog facility.
15. Processes BCP4 interrupts.

## 2.5 Interrupts

The interrupts for the ICU is as follows:-

| Number | Description |
|---|---|
| 0 | Power Down (cannot be masked or disabled)[1] |
| 1 | Machine Error (cannot be disabled) |
| 2 | Spare |
| 3 | Floating Point Overflow |
| 4 | Fixed Point Overflow |
| 5 | Executive Call (cannot be masked or disabled) |
| 6 | Floating Point Underflow |
| 7 | Timer A[4] |
| 8 | BCP4[2] |
| 9 | Timer B[3] |
| 10 | SSI[2] |
| 11 | Spare |
| 12 | Input/Output Level 1 |
| 13 | Instruction to User/RBI/LOSSN[2] |
| 14 | Input/Output Level 2 |
| 15 | Spare |

Notes.

1. Interrupt Number 0 has the highest priority. Priority decreases with increasing interrupt number.
2. All Interrupts are as per the 1750 standard except 8, 10 and 13. These ‘spare’ interrupts that have been allocated as above for the ICU.
3. Used by the Tartan Kernel to derive times for e.g. the `delay` facility – however, see section entitled “ICU Delay Adjustment”.
4. Used by the ICU code to produce a series of pulses to control the speed of the mechanisms (filter wheel and dichroic).

(The following is a summary of section 8.5.5.4 of the Tartan Compilation System Manual). There are five configurable interrupt masks that control the behaviour of the runtimes with respect to interrupts. They are defined in the linker control file `tlink17.lcf.` The purpose of each mask is as follows:

| | |
|---|---|
| ART_MASK | Used when executive space runtime code is executing |
| ARTELAB_MASK | Used when executive space main program is being elaborated |
| ARTTASK_MASK | Used when executive main program or user tasks are executing |
| PREEMPTER_MASK | Blocks all interrupts that might cause task pre-emption |
| CONNECT_MASK | Determines if a task entry may be directly connected to a particular hardware interrupt |

For both basic and operational code, the interrupts are enabled as follows:

| MASK | Value | Interrupt (1 = ENABLED) | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | (hex) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| ART_MASK | D5E4 | 1 | 1 | - | 1 | - | 1 | - | 1 | 1 | 1 | 1 | - | - | 1 | - | - |
| ARTELAB_MASK | DD40 | 1 | 1 | - | 1 | 1 | 1 | - | 1 | - | 1 | - | - | - | - | - | - |
| ARTTASK_MASK | DDE4 | 1 | 1 | - | 1 | 1 | 1 | - | 1 | 1 | 1 | 1 | - | - | - | - | - |
| PREEMPTOR_MASK | DC00 | 1 | 1 | - | 1 | 1 | 1 | - | - | - | - | - | - | - | - | - | - |
| CONNECT_MASK | 0100 | - | - | - | - | - | - | - | 1 | - | - | - | - | - | - | - | - |

## 2.6 ADA Exception Handling and Debugging

### 2.6.1 Overview

The package `DEBUG` provides a number of facilities which can be helpful in the event of an unexpected ADA exception or ‘hang’ of the code. In addition, a series of fixed memory locations (defined in package `MEMLOC`) are provided in which debugging information may be written. A combination of the two will usually indicate the problem area of the code.

### 2.6.2 Exception Handling

In the event of an ADA exception

1) a code of the form ‘Offset Code’ + ‘Exception Type’ is written to the reserved location FC0 (hex) – see next section. This is done using the procedure EXCEPTION_REPORT in package DEBUG. The ‘Offset Code’ indicates the package in which the exception occurred. The ‘Exception Type’ indicates which ADA exception occurred
2) A non-periodic engineering exception packet is issued containing two parameters. The first parameter contains an ‘Exception Code’ indicating which subsystem issued the exception. The second parameter is of the form ‘Offset Code’ + ‘Code Region’. The ‘Offset Code’ indicates which package was executing just prior to the exception. The ‘Code Region’ indicates within which region of package code the exception occurred. This second parameter is contained within reserved location FIRST_PROGRESS (see next section) of package DEBUG. Its contents are set up using calls to procedure PROGRESS in package DEBUG.

The Offset Codes (defined in package `DEBUG`) are as follows:

| Package | Offset Codes | |
|---|---|---|
| Name | Basic | Operational |
| ICU | 1000 | 2000 |
| CRC | 1100 | 2100 |
| DEMPSU | 1200 | 2200 |
| HK | 1300 | 2300 |
| MODEMAN | 1400 | 2400 |
| NHK | 1500 | 2500 |
| RBI | 1600 | 2600 |
| TASK_REPORT | 1700 | 2700 |
| TASKMAN | 1800 | 2800 |
| TC_Q | 1900 | 2900 |
| TC_VERIFY | 1A00 | 2A00 |
| TCQ | 1B00 | 2B00 |
| ICU_MEM_MANAGER | 1C00 | 2C00 |
| ICB | 1CB0 | 2CB0 |
| ICB_DRIVER | 1CBD | 2CBD |
| TM_Q | 1D00 | 2D00 |
| TMPSU | 1E00 | 2E00 |
| TMQ | 1F00 | 2F00 |
| MECHANISM | - | 4000 |
| DETDIGITAL | - | 4100 |
| TIME_MAN | 3200 | 4200 |
| MUTEX | 3400 | 4400 |
| HEATER | - | 4500 |
| MEM_MANAGER | 3C00 | 4C00 |
| SSI_DRIVER | 5500 | 6500 |
| SSI_IN | 5600 | 6600 |
| SSI_OUT | - | 6700 |
| SCIENCE_FM | - | 6800 |
| DPU | - | e000 |
| DPU_MEM_MANAGER | - | e100 |
| DETANALOG | - | E300 |
| MEMDPU | - | E400 |
| TM_MAN | 3500 | E500 |

The exception types are as follows:

| ADA Exception | Exception Type |
|---|---|
| Constraint Error | 0 |
| Program Error | 1 |
| Storage Error | 2 |
| Tasking Error | 3 |
| Other | 4 |

The exception codes are detailed in the telecommand and telemetry section of the User Manual (section 3.4.2 of XMM-OM/MSSL/ML/0010).

### 2.6.3 Reserved Locations

The package `MEMLOC` defines the following reserved locations for debugging purposes.

| Location Description | Notes | Address |
|---|---|---|
| FIRST_PROGRESS address | See exception handling above | 16#FC1#; |
| LAST_PROGRESS address | See section on `debug.adb` for description | 16#FC3#; |
| FIRST_EXCEPTION address | See exception handling above | 16#FC0#; |
| LAST_EXCEPTION address | See section on `debug.adb` for description | 16#FC2#; |
| SSI_ERROR_COUNT address | SSI Error Counter | 16#FC4#; |
| SSI_IN_BUF_PTR address | Pointer to next free location in SSI input buffer | 16#FC5#; |
| SSI_HEARTBEAT_COUNT address | SSI Heartbeat Counter | 16#FC5#; |
| SSI_INT_COUNT address | SSI Interrupt Counter | 16#FC6#; |
| BCP4_INT_COUNTER address | BCP4 Interrupt Counter | 16#FC7#; |
| RBI_INT_COUNTER address | RBI Interrupt Counter | 16#FC8#; |
| PROGRESS_SPECIAL address | Used as required to hold additional debug information | 16#FC9#; |
| PROGRESS_SPECIAL2 address | | 16#FCA#; |
| PROGRESS_SPECIAL3 address | | 16#FCB#; |
| PROGRESS_SPECIAL4 address | | 16#FCC#; |
| PROGRESS_SPECIAL5 address | | 16#FCD#; |
| PROGRESS_SPECIAL6 address | | 16#FCE#; |
| PROGRESS_SPECIAL7 address | | 16#FCF#; |

The following are reserved areas for counters for use by the named task (see section 2.4).

| Location Description | Address |
|---|---|
| LOAD_CENTROID_TABLE_TASK_COUNTER address | 16#FD0#; |
| LOAD_WINDOW_TABLE_TASK_COUNTER address | 16#FD1#; |
| HEARTBEAT_WATCHDOG_TASK_COUNTER address | 16#FD2#; |
| DATA_MANAGER_TASK_COUNTER address | 16#FD3#; |
| DPU_MEMORY_DUMP_TASK_COUNTER address | 16#FD4#; |
| CONTROL_TASK_COUNTER address | 16#FD5#; |
| PROCESS_TASK_COUNTER address | 16#FD6#; |
| TCQ_TASK_COUNTER address | 16#FD7#; |
| ICU_MEMORY_DUMP_TASK_COUNTER address | 16#FD8#; |
| MECH_TASK_COUNTER address | 16#FD9#; |
| ICU_TASK_COUNTER address | 16#FDA#; |
| WATCHDOG_TASK_COUNTER address | 16#FDB#; |
| PROT_TASK_COUNTER address | 16#FDC#; |
| BCP4_TASK_COUNTER address | 16#FDD#; |
| TC_Q_TASK_COUNTER address | 16#FDE#; |
| TIMER_A_TASK_COUNTER address | 16#FDF#; |

The following are reserved as code location indicators for the named task (see section 2.4).

| Location Description | Address |
|---|---|
| LOAD_CENTROID_TABLE_TASK_PROGRESS address | 16#FE0#; |
| LOAD_WINDOW_TABLE_TASK_PROGRESS address | 16#FE1#; |
| HEARTBEAT_WATCHDOG_TASK_PROGRESS address | 16#FE2#; |
| DATA_MANAGER_TASK_PROGRESS address | 16#FE3#; |
| DPU_MEMORY_DUMP_TASK_PROGRESS address | 16#FE4#; |
| CONTROL_TASK_PROGRESS address | 16#FE5#; |
| PROCESS_TASK_PROGRESS address | 16#FE6#; |
| TCQ_TASK_PROGRESS address | 16#FE7#; |
| ICU_MEMORY_DUMP_TASK_PROGRESS address | 16#FE8#; |
| MECH_TASK_PROGRESS address | 16#FE9#; |
| ICU_TASK_PROGRESS address | 16#FEA#; |
| WATCHDOG_TASK_PROGRESS address | 16#FEB#; |
| PROT_TASK_PROGRESS address | 16#FEC#; |
| BCP4_TASK_PROGRESS address | 16#FED#; |
| TC_Q_TASK_PROGRESS address | 16#FEE#; |
| TIMER_A_TASK_PROGRESS address | 16#FEF#; |
| | |

## 2.7 ICU Delay Adjustment :

The standard 31750 processor specification requires a timer B clock of 100 khz. However, for the RGS and OM (same processors design), a 62.5 khz clock is used. As this timer is used to control any requested delays, this will produce an error factor of 0.625 in the `delay` statements. For example a delay of 1 second will produce an actual delay of (1/0.625 second). In the majority of cases, this does not matter as the delay is used for tasks de-scheduling. However in other cases (such as the HK timer - default to 10 seconds), a correction factor of 0.625 is applied to get an accurate delay. For example a more accurate 10 second delay can be achieved as follows :

```
TIME_CORRECTION = 0.625

delay (10.0 * TIME_CORRECTION).
```

However, for the OM code, the Tartan run time library (`madart.tlib`) was modified so that the above correction is automatically applied by modifying the handling of Timer B interrupts.

# 3. Design Method

The design methodology is “object-based”, i.e. most modules in the system denote an object identified in the system (e.g. ‘real’ objects such as the detector, or ‘soft’ objects such as an on-board telemetry manager). These objects were then implemented in the ADA language.

By object, we mean an entity that has
1. a state ( i.e. a value or values)
2. actions it suffers or can apply to other objects

It is recognised that it is not possible to have a perfect knowledge of a software problem at the start, i.e. the growth of understanding is an iterative one. It is therefore assumed that previously unrecognised problems will be found as the project proceeds. However, since the design corresponds to the ‘real world’, it is hoped that the resultant changes will not radically affect the design. Instead, they will only impact those modules associated with the affected objects. This is also a good system in situations where important sub-systems require early development, and where some elements are not well known early on.

Therefore, the steps used are as follows:
- Identify the objects and their attributes.
- Identify the operations that affect each object and the operations that each object must initiate.
- Establish the visibility of each object in relation to other objects.
- Establish the interface of each object.
- Implement each object.

There are also a number of modules which are of a ‘library’ nature - i.e. they provide constants, definitions and routines of a general nature with no specific object in mind.

The design aim is that modules are implemented as ADA packages. However, when speed of execution is a requirement, the language of choice is 1750 Assembler.

We have also been guided by ESA PSS-05 software engineering standards, though some differences arise because of the small in-house software team (2-3 people) involved in this project and the prioritised nature of the work required.

# 4. ADA Overview

## *4.1 Basic Definitions*

A full description of the ADA language is beyond the scope of this document, but the following summary of some ADA features may be useful in understanding the descriptions in the following sections.

**Program Units** - An ADA program is composed of one or more program units. It is standard OM practice to compile these separately. Program Units consist of Subprograms, Task, Packages and Generic Units. All ADA program units have a similar two part structure, consisting of a Specification and a Body. These are also compiled separately

**Specifications** identify the information visible to a client (i.e. the caller) of that program unit.

**Bodies** contain the implementation details and will be hidden from the client.

**Subprograms** are either Procedures or Functions and express a sequential action. A function is the same as a procedure, except that its primary purpose is to return a calculated value. (A **Main Program** is a special case of a Subprogram that is called directly when the code starts running. It can also be regarded as a separately running task)

**Tasks** defines an action that is executed in parallel with other tasks.

**Packages** are a collection of computational resources, encapsulating data types or instances thereof, subprograms, tasks or other packages.

**Generic Units** are templates for subprograms and packages and serve as the primary mechanism for building reusable software components.

The following table summarises the above characteristics and, additionally, lists the applications for each.

| Program Unit | Characteristic | Applications |
|---|---|---|
| Subprogram | Sequential Action | Main Program Unit<br>Definition of Functional Control<br>Definition of Type Operations |
| Package | Collection of Resources | Named Collection of Declarations<br>Groups of Related Program Units<br>Abstract Data Type<br>Abstract State Machines<br>Objects |
| Task | Parallel Action | Concurrent Actions<br>Routing Messages<br>Controlling Resources<br>Interrupts |
| Generic Unit | Template | Reusable S/W Components |

## 4.2 Task Scheduling

At a given time during the execution of an ADA program, there are a set of tasks that are eligible for execution. The process of choosing a subset of tasks to actually run is called scheduling. The code uses the Tartan implementation of this process.

Scheduling has two parts:

1. Scheduler
2. Dispatcher

The **scheduler** is responsible for:

- adding a task to the set of executable tasks
- removing a task from the set of executable tasks
- selecting a task from the set to be the next task executed

The scheduler is implemented as a strictly priority ordered queue. The task with the highest priority is selected for execution. New tasks are inserted after tasks of equal priority within priority levels.

When the **dispatcher** is invoked, it causes the currently executing task to be suspended and replaced by the next task selected by the scheduler. If the current task is also the task selected to be the next by the scheduler, no change occurs.

The dispatcher is invoked whenever an ADA tasking or `delay` operation causes execution of the current task to be blocked, or when a high priority task becomes ready (for instance, by the expiration of a `delay`) and pre-empts a task with lower priority. This method of pre-emptive scheduling minimises the amount of time a high-priority task must wait for execution after a `delay` operation.

## 4.3 Identifier Naming Conventions

In common with standard ADA coding practice, `lower case` letters indicate reserved words, `UPPER CASE` letters indicate identifiers.

## 4.4 Programming Standards

Ref - 9 defines the ADA coding standards used in the project.

# 5. Components

## *5.1 Overview*

The components are grouped into the following categories:

1. Main Program.
2. Specifications and bodies of Named 'objects'.
3. Specifications and bodies of Libraries of Related Routines that do *not* form an object (e.g. a maths library).
4. Specifications of General Definitions e.g. Types and Constants.
5. Miscellaneous

## *5.2 File Naming Conventions*

| | |
|---|---|
| `filename.ada` | - a file containing a 'main' program |
| `filename.ads` | - a file containing an ADA specification |
| `filename.adb` | - a file containing an ADA body |
| `filename.adp` | - a file containing both ADA specification *and* body |
| `filename.asm` | - a file containing MSSL written 31750 assembler code |
| `filename.ASM` | - a file containing Tartan supplied, MSSL modified, assembler code |
| `filename.ad[sb]` | - refers to both `filename.ads` and `filename.adb` |

The `filename` for a specification (`.ads`) is the same as the corresponding package body (`.adb`) name.

`filename` is always the name of the package specification and/or body or main program included in the file.

## 5.3 Component Summary

### 5.3.1 Objects

The following table indicates how the principal s/w objects, as indicated on the overview diagrams given earlier, 'map' to files. This table is valid for both basic and operational mode code provided, of course, that the named object is present.

| Object | Consist of Filename(s) | Description |
|---|---|---|
| DEMPSU | dempsu.ad[sb] | Routines to control the DEMPSU |
| DETECTOR | detanalog.ad[sb] | Routines to control and monitor the analog functions of the detector electronics, can issue safing command. |
| | detdigital.ads | Routines to control and monitor the digital functions of the detector electronics |
| | detector.ads | Provides single interface for all detector routines |
| DPU CONTROLLER | dpu.ad[sb] | Sends commands and receives data from the DPU. |
| | dpu_mnemo.ads | Provides DPU Mnemonic definitions |
| HK | hk.ad[sb] | Obtains HK items, constructs the HK packet and sends result to the TM queue. |
| ICB | icb.ad[sb] | Controls access to the ICB interface code in icb_driver. |
| | icb_driver.ad[sb] | Provides the routines to perform I/O on the ICB interface |
| MECHANISMS | mechanism.ad[sb] | Code to control the filter wheel and dichroic, can issue safing command, and request heater settings. |
| | timer_a_ih.ad[sb] | Routines to handle timer A interrupts. |
| MEMORY MANAGER | dpu_mem_manager.ad[sb] | Routines to perform DPU memory load and dump |
| | icu_mem_manager.ad[sb] | Routines to perform ICU memory load and dump |
| | mem_manager.ad[sb] | Interprets memory management packet and call appropriate routine |
| | peek_poke.ads | Examines addresses in various memory modes. |
| | peek_poke.asm | |
| | memdpu.ad[sb] | Intercepts DPU memory dumps (part of dpu_mem_manager), constructs appropriate packet and places it in TM queue |
| RBI | bcp4_ih.ads | Routines to handle BCP4 interrupt |
| | bcp4_ih.asm | |
| | rbi.ad[sb] | Routines to perform non-interrupt driven RBI functions |
| | rbi_ih.ads | Routine to handle RBI interrupt |
| | rbi_ih.asm | |
| SSI | ssi_driver.ad[sb] | Routines to handle non interrupt part of SSI I/O |
| | ssi_ih.ads | Routines to handle SSI interrupts |
| | ssi_ih.asm | |
| | ssi_in.ad[sb] | Constructs DPU data block sent over SSI |
| | ssi_out.ad[sb] | Sends a DPU command over the SSI |

Objects (continued)

| Object | Consist of Filename(s) | Description |
|---|---|---|
| TASK MANAGER | modeman.ad[sb] | Performs Mode switching |
| | taskman.ad[sb] | Interprets Task Management packets and call appropriate routine |
| | reset.ads | Called by modeman when switching from basic to operational. |
| | reset.asm | |
| TC PROCESS | icu.ada | The main program. Takes valid TC packets and distributes them to the appropriate Packet Manager |
| | tc_q.ad[sb] | Routines to manipulate TC packets in the queue |
| | tc_verify.ad[sb] | Routines to validates TC packets |
| | tcq.ad[sb] | Controls access to routines to extract packets from TC queue |
| THERMAL | heater.ad[sb] | Routines to control heaters |
| TIME MANAGER | time_man.ad[sb] | Interprets time management packets and call appropriate routines |
| TM QUEUE | tm_q.ad[sb] | Routine to manipulate the TM packets, issues safing if TM queue full for > 1 min. |
| | tmq.ad[sb] | Controls access to routines to manipulate TM packets |
| | nhk.ad[sb] | Constructs non periodic HK packets and place them in TM queue |
| | science_fm.ad[sb] | Constructs Science Packets and places them in the TM queue |
| | task_report.ad[sb] | Constructs task report packets an places them in the TM queue |
| TMPSU | tmpsu.ad[sb] | Routines for low level TMPSU control and monitoring |

### 5.3.2 Definitions

The following table indicate which files are used for definitions only. They contain only specifications. There are no corresponding files containing a package body.

| **Filename(s)** | **Description** |
|---|---|
| `types.ads` | Defines additional ADA types |
| `packet.ads` | Defines the packet structure |
| `importance.ads` | Defines ADA task priorities |
| `memloc.ads` | Defines key memory locations |
| `INTVEC.ASM` | Linkage and Service Pointers for Interrupts |
| `USERDEFS.ASM` | Timer correction factor + ADA run time constants and masks |

### 5.3.3 Library Routines

The following table contain routines that do not map to a single object but are instead used by many. They are therefore classed as ‘libraries’.

| **Filename(s)** | **Descriptions** |
|---|---|
| `crc.ad[sb]` | CRC calculation routines |
| `debug.ad[sb]` | Debugging utility routines |
| `mutex.ad[sb]` | Used to provide mutually exclusive access to various resources |

# 6. Detailed Component Description

## 6.1 Introduction

This section contains:

1) A detailed description of the Bootstrap code.
2) A components section describing the software components that make up the Basic and Operational code.This components section is subdivided according to the type of the components contained within the specified file e.g. Main Program, Package representing an Object, Packages representing a Library etc. It has been compiled by extracting 'flagged' comments of a design nature - e.g. 'Structured English' descriptions - from the code itself.

### 6.1.1 ADA Procedure and Function Notation

Many of the components described in this section assume knowledge of the calling convention, or interface, to procedures and functions in ADA. A summary is therefore presented here.

The list of parameters in a subprogram or function call are known as *actual* parameters; inside the subprogram they are called *formal* parameters. They are passed in one of 3 calling modes:

- `in` Only the actual value is used; the subprogram cannot modify the value.
- `out` The subprogram creates a value but does not use the value of the actual parameter
- `in out` The subprogram uses the value from the actual parameter and may assign a new value to it.

If omitted, `in` mode is the default.

The notation used in subprogram and function calls is as follows, e.g.:

```
procedure COUNT_LEAVES_ON_BINARY_TREE;
```

illustrates an interface to a procedure with no arguments.

```
procedure ROTATE
     (POINT    : in out  TRANSFORM.COORDINATE;
      ANGLE    : in           UNITS.RADIANS);
```

states that procedure `ROTATE`s formal first argument has the name `POINT`, is of type `COORDINATE` (which in turn is defined in the package `TRANSFORM`) and that its calling mode is `in out`. Similarly the second argument has the name `ANGLE`, is of type `RADIANS` (defined in package `UNITS`) and that its calling mode is `in`.

Similarly for functions:

```
function COS
     (ANGLE         : in      UNITS.RADIANS) return FLOAT;
```

shows that the function `COS` returns the predefined type `FLOAT`.
**Note** that function arguments can only be of mode `in`.

### 6.1.2 ADA Task and Entry Notation

A task specification introduces the name of the task, together with any entry points to the task. It thus defines the communication paths (entries) available to other tasks. It may also define the priority at which a task runs. An entry declaration has a form similar to a subprogram specification. e.g:

```
task PROTECTED_STACK is
  pragma priority(100);             -- define priority
  entry POP  (ELEMENT : out INTEGER);    -- define entry point
  entry PUSH(ELEMENT : in INTEGER);      -- define entry point
end PROTECTED_STACK;
```

It is always necessary to prefix entry calls with the task name. e.g.:

```
PROTECTED_STACK.POP(MY_VALUE);
```

Occasionally, tasks are specified as task types. This is done if there is more than one instance of the task or to allow certain ADA pragmas (special instructions to the compiler) to be obeyed that are only supported under task types. The following is equivalent to the above specification.

```
task type PROTECTED_STACK_TYPE is  -- define the type.
  pragma priority(100);             -- define priority
  entry POP  (ELEMENT : out INTEGER);    -- define entry point
  entry PUSH(ELEMENT : in INTEGER);      -- define entry point
end PROTECTED_STACK_TYPE;

PROTECTED_STACK : PROTECTED_STACK_TYPE;                          -- create an
instance of the task.
```

### 6.1.3 Use of ADA Pragma's

A pragma is a statement that conveys information to the compiler. The following ADA pragmas are used throughout the code. The following is a summary of the usage.

#### *6.1.3.1 ELABORATE*

Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause.

This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit.

#### *6.1.3.2 FOREIGN_BODY*

This provides a way to access entities written in languages other than ADA. It must appear in the visible part of the package before any declarations – see section 4.1.2.2 of the Tartan ADA Compilation System Manual.

It dictates that all subprograms and objects in the package are provided by means of a foreign object module. In the case of the ICU, the language used is always assembler and therefore the pragma takes the argument string "ASM".

#### *6.1.3.3 INLINE*

Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible.

#### *6.1.3.4 LINKAGE_NAME*

This pragma associates an ADA entity (e.g. subprogram or variable name) with a text string meaningful externally to, say, a linkage editor. It is usually used to equate that entity to its equivalent in assembler code.

#### *6.1.3.5 OPTIMIZE*

Takes one of the identifiers `TIME` or `SPACE` as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part.

It specifies whether time or space is the primary optimisation criterion.

#### *6.1.3.6 PACK*

Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause.

The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type.

#### *6.1.3.7 PRIORITY*

Takes a static expression of the predefined integer subtype `PRIORITY` as the single argument. This pragma is only allowed within the specification of a task unit or immediately within the outermost declarative part of a main program.

It specifies the priority of the task (or tasks of the task type) or the priority of the main program. **N.B.** The package `IMPORTANCE` defines of all task priorities used as arguments to this pragma.

#### *6.1.3.8 SHARED*

Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification.

This pragma specifies that every read or update of the variable is a synchronization point for that variable i.e. no optimisation is performed which might lead to the value contained in the variable not always being up-to-date.

#### *6.1.3.9 SUPPRESS*

Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit.

The identifier is that of the check that can be omitted. The name (if present) must be either a simple name or an expanded name and it must denote either an object, a type or subtype, a task unit, or a generic unit; alternatively the name can be a subprogram name, in which case it can stand for several visible overloaded subprograms.

The following checks correspond to situations in which the exception `CONSTRAINT_ERROR` may be raised; for these checks, the name (if present) must denote either an object or a type.

- `ACCESS_CHECK:` When accessing a selected component, an indexed component, a slice, or an attribute, of an object designated by an access value, check that the access value is not null.

- `DISCRIMINANT_CHECK:` Check that a discriminant of a composite value has the value imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.

- `INDEX_CHECK:` Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.

- `LENGTH_CHECK:` Check that there is a matching component for each component of an array, in the case of array assignments, type conversions, and logical operators for arrays of boolean components.

- `RANGE_CHECK:` Check that a value satisfies a range constraint. Also, for the elaboration of a subtype indication, check that the constraint (if present) is compatible with the type mark. Also, for an aggregate, check that an index or discriminant value belongs to the corresponding subtype. Finally, check for any constraint checks performed by a generic instantiation.

The following checks correspond to situations in which the exception `NUMERIC_ERROR` is raised. The only allowed names in the corresponding pragmas are names of numeric types.

- `DIVISION_CHECK:` Check that the second operand is not zero for the operations `/`, `rem` and `mod`.

- `OVERFLOW_CHECK:` Check that the result of a numeric operation does not overflow.

The following check corresponds to situations in which the exception `PROGRAM_ERROR` is raised. The only allowed names in the corresponding pragmas are names denoting task units, generic units, or subprograms.

- `ELABORATION_CHECK:` When either a subprogram is called, a task activation is accomplished, or a generic instantiation is elaborated, check that the body of the corresponding unit has already been elaborated.

The following check corresponds to situations in which the exception `STORAGE_ERROR` is raised. The only allowed names in the corresponding pragmas are names denoting access types, task units, or subprograms.

- `STORAGE_CHECK:` Check that execution of an allocator does not require more space than is available for a collection. Check that the space available for a task or subprogram has not been exceeded.

## 6.2 Bootstrap Code

### 6.2.1 Introduction

The OM Bootstrap resides in the ICU memory, and is the first piece of code to be executed by the ICU processor after a reset or power up. It's purpose is to initialise the instrument hardware and higher level software.

This code is blown into PROM and hence it will not be able to be changed after launch.

The ICU is designed to have 16K words of PROM, each PROM chip holds 8K octets of code. The 16K words available will hold both the Bootstrap code and the Basic mode code.

The PROM's to be used for this are very higher power ones, so the on time of these chips needs to minimised.

The Bootstrap code will be initiated by three possible alternatives (See Figure 1) :-

1. Power On or the Main Power Bus to the instrument has been interrupted.
2. A RESET ICU, warm or cold start, command has been received by the instrument from the spacecraft.
3. The ICU watchdog has timed out. (The RBI's watchdog timer is to be used for this function. [App-2]).

Since the ICU uses an MA31750 processor, 31750 assembler language is a natural choice to be adopted for the Bootstrap code implementation.

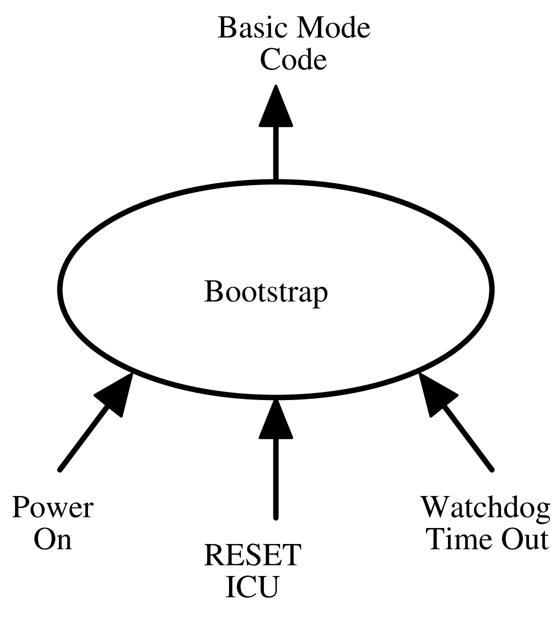



**Figure 1 ICU Bootstrap Operational Architecture**

### 6.2.2 BOOTSTRAP FUNCTIONALITY

This section specifies what the Bootstrap code does, and the implementation and operational constraints.

Essential (Priority 1) :

R1.1 Enable the spacecraft OBDH to be able to perform read and write operations to the ICU RAM memory. ([App-1] R4.1.3.3.1-4). (This is required in order that the spacecraft can send telecommands to, and take telemetry data from the ICU RAM, even though these will not be supported at this time. This also allows spacecraft low level access to the ICU RAM in case of problems, e.g. patch access.)
R1.2 Safe the instrument as it is possible for the ICU to remain in reset mode for an indefinite time.
R1.3 Load all PROM code into RAM, turn off PROM, and run Bootstrap code from RAM.
R1.4 Initialise all interrupt handlers to return to caller.
R1.5 Follow OBDH protocol to next mode.

Highly Desirable (Priority 2) :

R2.1 Perform RAM checks relevant to loading code from ROM to RAM and report any errors to ground in RBI software indication field. (Checks may be pre- or/and post- loading) (Is not classed as essential as RAMs used are very SEU immune)
R2.2 In event of a RAM error, provide means to avoid bad RAM locations.
R2.3 Unused interrupt handlers to store count of times called to be reported in housekeeping for diagnostic purposes.

Constraints :

C1 Minimise time that the ROM is powered. (To less than 100 msecs for nominal operations, to avoid overheating of components, to minimise total power consumed and to avoid brown outs).
C2 Bootstrap + Basic code must fit within 16K 16-bit words. (Bootstrap code must fit within 2K words baseline allocation.)
C3 Implement Bootstrap on an MA31750 processor operating at 8Mhz, with (TBD) PROM.
C4 Comply with OBDH protocols (for next mode) [App-1].
C5 On entry to Bootstrap the ICU hardware status will be :-
_
- Interrupts are disabled.
- DMA by the RBI will be disabled.
- The PROMs will be powered on.

### 6.2.3 BOOTSTRAP IMPLEMENTATION

This section specifies how the Bootstrap code is implemented.

**S0. The Bootstrap Code will be implemented in 31750 assembler on an MA31750 processor.**

- Since the ICU uses an MA31750 processor, 31750 assembler is a natural choice to be adopted for the ICU Bootstrap code language.

- The use of assembler is also consistent with stringent memory and speed limitations consistent with these Bootstrap requirements.

The following specifies what the Bootstrap will do once it is invoked, in chronological order :-

**S1. Enable Spacecraft to have read & write access into the instrument's memory.**

This is required to :-
- Enable the spacecraft to be able to send telecommands to the ICU.
- Enable the spacecraft to take telemetry data from the ICU.
- Allow spacecraft read/write access to the ICU RAM in case of problems.

**(a)** Enable DMA: The Bootstrap will write any value to IO location DMAE, defined in 31750 assembler.

(b) Command RBI into Reset state: The Bootstrap will write 8000 hex to the RBI Configuration register, IO address 6806 hex.(Note that if a watchdog time-out has occurred this command will have no effect. The value in the RBI Status register, bits 0-3, seen by the Ground System will be zero.)

(c) Write the CCA address into RBI's base address register.

(d) Send reset page address command to RBI configuration register.

**S2. Copy code in PROM to RAM, turn off PROM and run in Bootstrap code from RAM.**

– Copy bootstrap code from ROM into RAM.
– Options:
(a) Immediately turn of PROM and go to S3, or
(b) Check that RAM code is OK and/or do checksum on code copied.
    – If OK, then turn off PROM, run code from RAM, and go to S3.
    – If not OK, then try again.
    – If not OK for a second time, find four words in consecutive RAM memory which are OK and copy two jump instructions to RAM which loop to each other.
        – A value is written to the RBI Status register bits 12-15, Software Indication field, to indicate to the Ground System the Bootstrap failed.
        – The PROM is then turned off and the two jump instructions are run.
        (The Ground System then can then study the problem. If sufficient code can be loaded using Low Level DMA commands under Ground System control the Ground System can then change one of the addresses in the jump instructions so control is passed to the loaded code.)
    – If four words in consecutive RAM memory cannot be found which are OK :-
    – Write a value to the RBI Status register to indicate to the Ground System that total RAM failure has occurred.
    – Turn off the PROM.
    – Copy Basic mode code from PROM into RAM unless the bootstrap was started due to a reset ICU no copy interrogation to the RBI.

**S3. Safe various components of the instrument.**

This step will always turn off the heaters and the filter wheel phases, reset the DPU and move the filter wheel to the blocked position using the coarse sensor only. Additionally, the high voltage unit and the TMPSU secondaries will be turned off if the bootstrap is running due to a "warm start" (i.e. ICU code has already been running).

**S4 Initialise all interrupt handlers to return to caller.**

This is achieved at code assembly time.

**S5. Determine next mode as per OBDH Bus Protocol Specification. [1]**

Sections of [1] that describe this procedure are figure 4.1.3-1 and R4.1.3.3.1-1.

A pseudo code listing of this procedure is given below :-

```
    - Copy the bootstrap to RAM
    - Read RBI configuration register
      If WD bit set                          - If watchdog time-out has occurred
        Set jump pointer to WD_ENTRY
        Set boot type flag to watchdog
        Goto PROM_OFF
      End If
    - Read RBI instruction to RBI register
      If 0000 XXXX 0000 0000                 - Reset ICU Cold Start command)
        Set jump pointer to BASIC_START
        Set boot type flag to cold
      Else
        Set jump pointer to READ_LOOP
        Set boot type flag to warm
        If 0000 XXXX 0101 1110            - Reset ICU Warm Start command, no copy.
            Goto PROM_OFF
        End if
      End if                                 - Reset ICU Warm Start command, copy.
    - Copy Basic mode code into RAM          - Fall through

PROM_OFF:
        Turn off PROM
        Perform safing
        Use jump pointer to goto to next procedure

READ_LOOP:
        Read RBI configuration register.
            If IT1 bit not set goto READ_LOOP
            End If
            If IT1 bit is set read Instruction-to-RBI Register
                If 1111 XXXX 0000 0000          - Start ICU command.
                    Goto BASIC_START
                Else if 0000 XXXX 0101 XXXX     - Reset ICU Warm Start Cmd
                    Restart Bootstrap
                End If
            Else goto READ_LOOP
            End if
```

BASIC_START:

Command RBI into Running State
Transfer control to Basic mode Code

WD_ENTRY:

Goto WD_ENTRY - *wait for reset command from*
- *ground and then go back to S1.*

### 6.2.4 Design and Implementation

The XMM-OM bootstrap is based on a modified version of the Tartan supplied Adascope kernel.

Due to the power requirements of the PROMs the bootstrap must copy itself to code space RAM, turn off the PROMs and continue running from RAM. Additionally, before shutting off the PROMS the bootstrap must decide whether to copy out the Basic mode code too.

Flow charts of the bootstrap are shown in figures 2 to 6 and the corresponding pseudo code is given below.

```
    - Copy the bootstrap to RAM
    - Read RBI configuration register
      If WD bit set                              - If watchdog time-out has occurred
        Set jump pointer to WD_ENTRY
        Set boot type flag to watchdog
        Goto PROM_OFF
      End If
    - Read RBI instruction to RBI register
      If 0000 XXXX 0000 0000                     - Reset ICU Cold Start command)
        Set jump pointer to BASIC_START
        Set boot type flag to cold
      Else
        Set jump pointer to READ_LOOP
        Set boot type flag to warm
        If 0000 XXXX 0101 1110               - Reset ICU Warm Start command, no copy.
            Goto PROM_OFF
        End if
      End if                                     - Reset ICU Warm Start command, copy.
    - Copy Basic mode code into RAM              - Fall through

PROM_OFF:
            Turn off PROM
            Perform safing
            Use jump pointer to goto to next procedure

READ_LOOP:
            Read RBI configuration register.
                    If IT1 bit not set goto READ_LOOP
                    End If
                    If IT1 bit is set read Instruction-to-RBI Register
                        If 1111 XXXX 0000 0000             - Start ICU command.
                            Goto BASIC_START
                        Else if 0000 XXXX 0101 XXXX        - Reset ICU Warm Start Cmd
                            Restart Bootstrap
                        End If
                    Else goto READ_LOOP
                    End if

BASIC_START:
```

```
        Command RBI into Running State
        Transfer control to Basic mode Code

WD_ENTRY:
        Goto WD_ENTRY   - wait for reset command from
                        - ground and then go back to S1.


Perform safing

    if boot type flag = cold
      if Blue Processing Electronics is on
        goto WARM_SAFE
      end if
    call WARM_SAFE1                         - common safing procedure with warm start
    call SAFE_FW                            - safe the filter wheel
    Use jump pointer to goto to next procedure    - will be the read loop for go command
                                            - in the case of a warm start, or Basic mode
                                            - code in the case of a cold start
    end if


WARM_SAFE
    turn off high voltages
    call WARM_SAFE1                         - common safing procedure with cold start
    pause for 5 seconds
    turn off secondary voltages
    call SAFE_FW procedure
    Use jump pointer to goto to next procedure    - will be the read loop for go command
                                            - in the case of a warm start, or Basic mode
                                            - code in the case of a cold start.

WARM_SAFE1
    turn off all heaters
    turn off filter wheel phases
    reset the DPU
    return from sub-procedure

SAFE_FW
    initialise counters, phase variable, etc.
    turn on coarse sensor LED
    if command failed
      goto EXIT
    end if
NEXT
    do
      calculate next phase
      read coarse sensor
      if read failed
        goto EXIT
      end if
```

```
        if coarse sensor detected
          goto COARSE_SEEN
        else
          set coarse counter            - to detect when we see coarse for second time
        end if
        energise next phase
        if command failed
          goto EXIT
        end if
        delay for time specified in acceleration table
        decrement step counter
      while step counter > 0
EXIT
      save results of procedure
      turn off LED and phases
      return from sub-procedure

COARSE_SEEN
      if coarse counter = 0             - then gap between seeing coarse sensor
        goto NEXT
      end if
      increment coarse counter
      if coarse counter = 2             - seeing coarse for second time
        store step counter in steps remaining location
        set step counter to 1257/1258 (redundant/prime)        - steps needed until in blocked position
        goto NEXT
      else
        goto NEXT
      end if
```

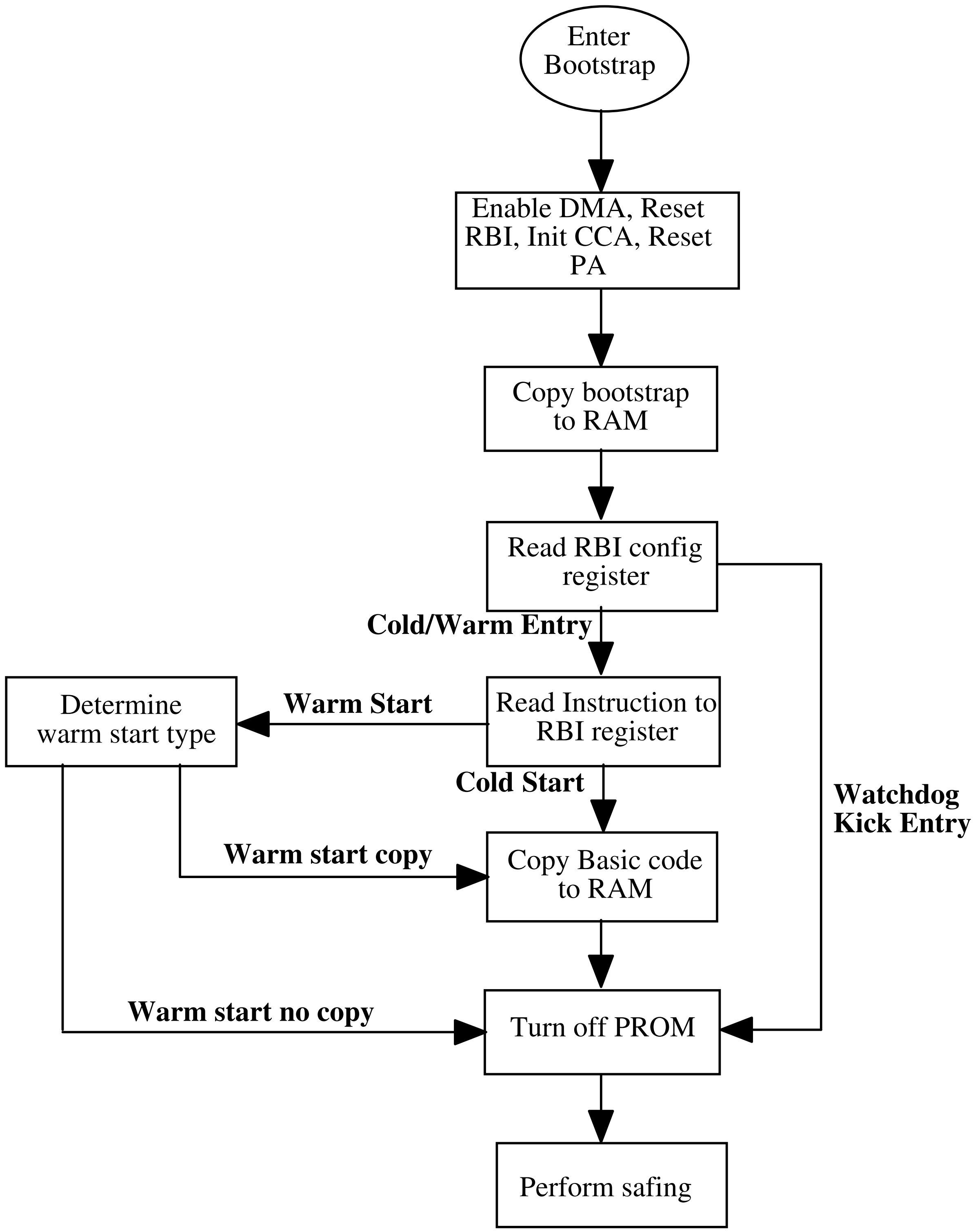



**Figure 2 Determination Of Boot Type**

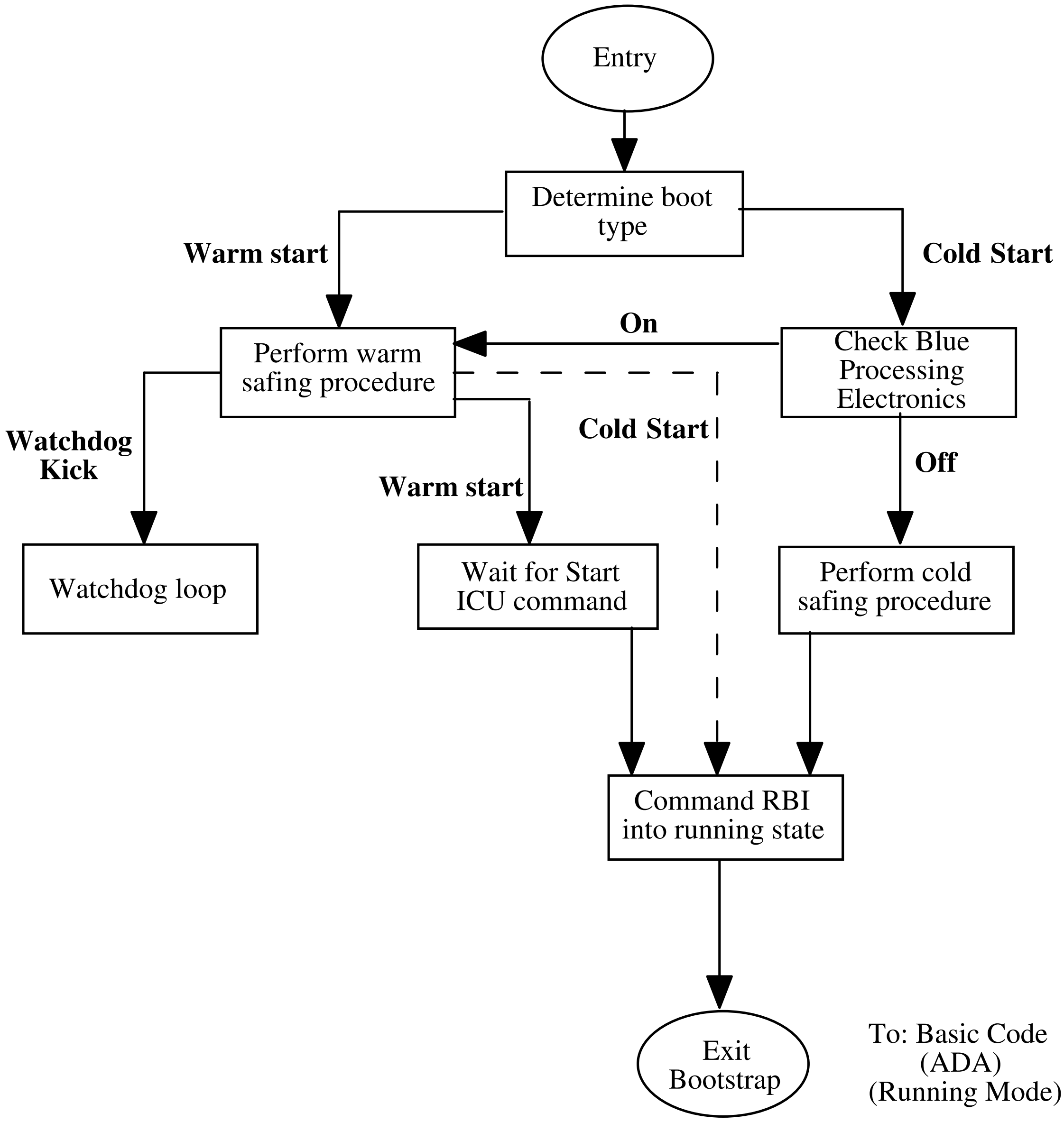



**Figure 3 Perform Safing**

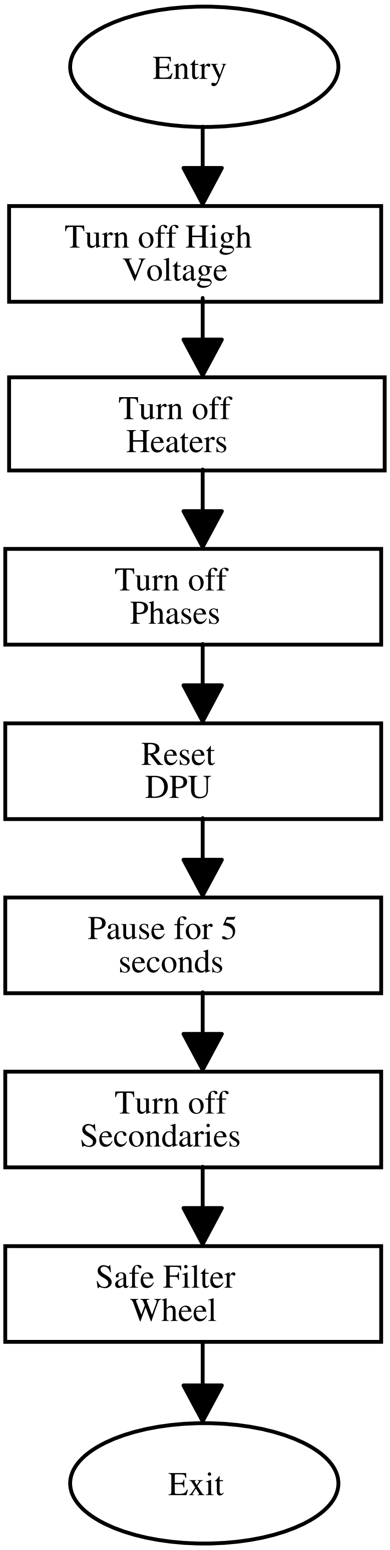



**Figure 4 Warm Safing Procedure**

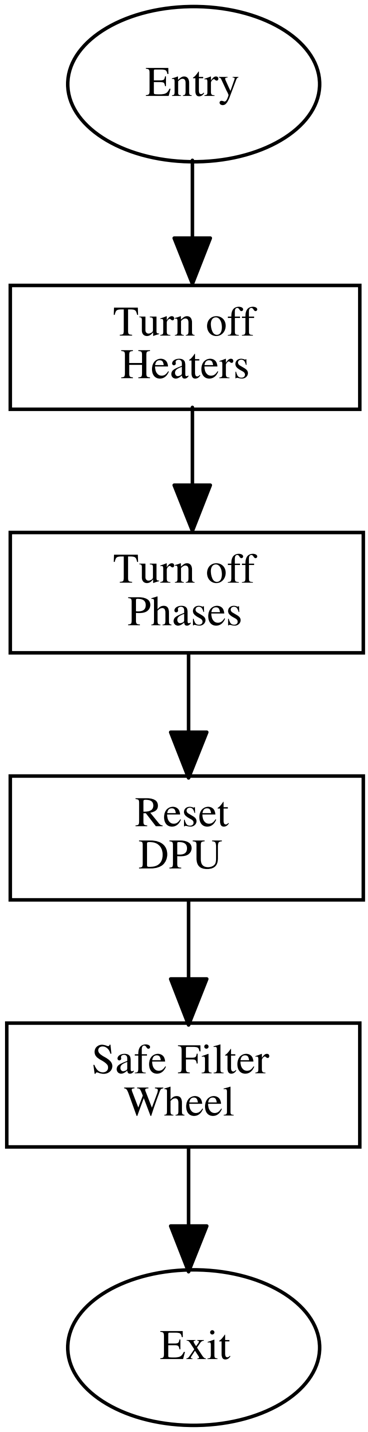



**Figure 5 Cold Safing Procedure**

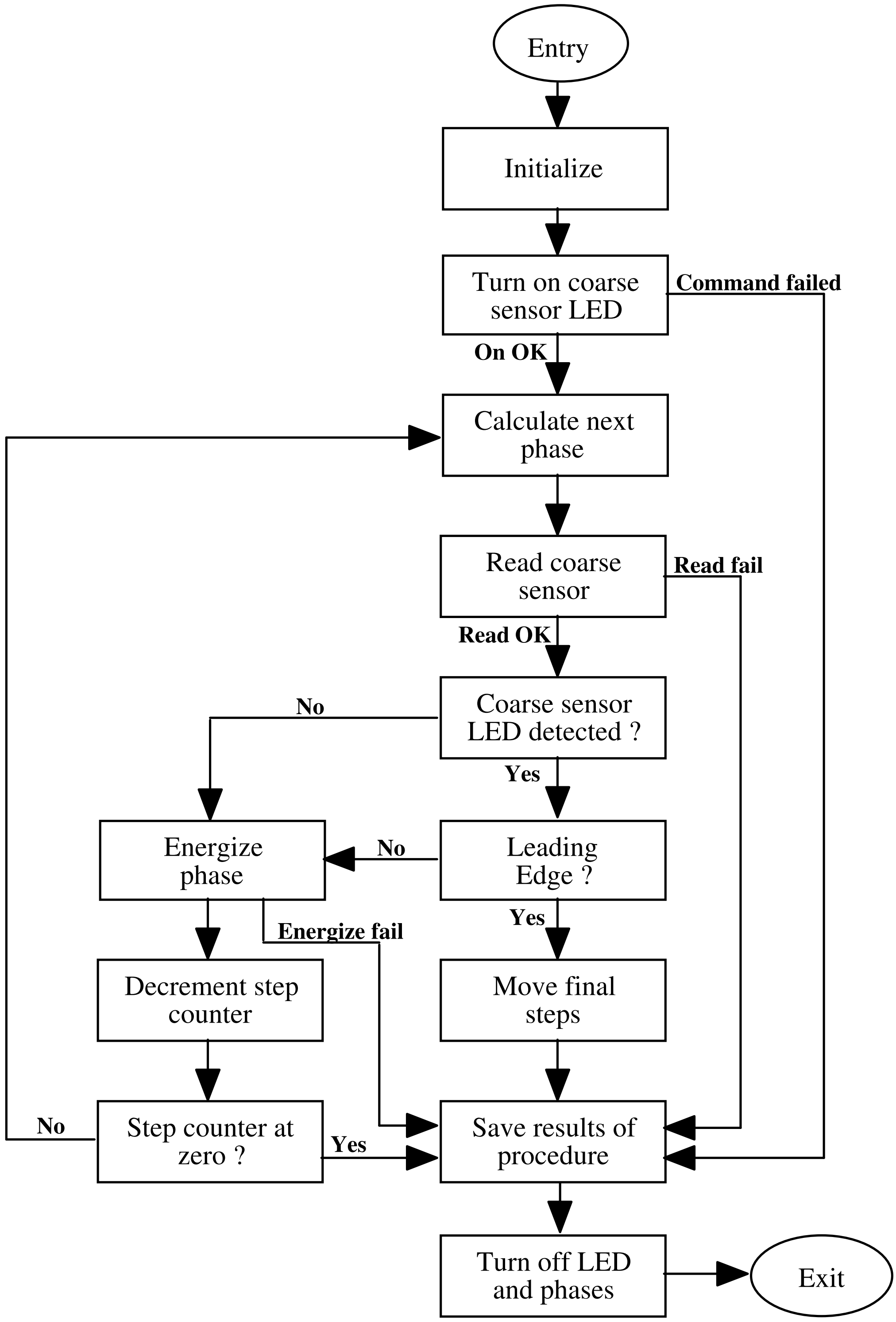



**Figure 6 Safe Filter Wheel Procedure**

### 6.2.5 Variables

This section lists the variables which are used by the bootstrap. These variables are also available for the Basic mode code to inspect and downlink to the ground, providing information as to what happened during the bootstrap process. They appear in the linker map reference as `BOOTSTRAP_PARAMS.` The length of each parameter is given in 16-bit words and the address is in hexadecimal. These variables are to be found in data space. All the variables are defined in the file `emboot.asm`.

| Address | Length | Parameter | Description |
|---|---|---|---|
| 03FE | 1 | PRM_RED | Indicates whether the Primary or Redundant system is running<br>1 = Primary, 0 = Redundant. |
| 03FF | 1 | VERSION | Version number of the Bootstrap code only. Constant = 0137. |
| 02C7 | 1 | BOOT_TYPE | Boot type detected by the Bootstrap.<br>0 = Power up<br>1 = Reset cold<br>2 = Reset warm copy<br>3 = Reset warm no copy<br>4 = Watchdog kick |
| 02C8 | 1 | PROGRESS | Bit pattern recording the safing procedures completed. See note 1. |
| 02C9 | 1 | ICB_GOOD | Counter recording the total number of valid ICB commands sent. Range [0,FFFF] |
| 02CA | 1 | ICB_SYNC_ERRS | Counter recording the total number of ICB sync errors detected. Range [0,FFFF] |
| 02CB | 1 | ICB_EXT_ERRS | Counter recording the total number of ICB extension errors detected. Range [0,FFFF] |
| 02CC | 1 | ICB_TX_ERRS | Counter recording the total number of ICB transmission errors detected. Range [0,FFFF] |
| 02CD | 1 | ICB_TIMEOUT | Counter recording the total number of ICB time-out errors detected. Range [0,FFFF] |
| 02CE | 1 | ICB_DEAD | 0 = Alive, 1 = Dead |
| 02CF | 1 | COARSE_SEEN | Indicates whether the Coarse Sensor was detected when safing the Filter Wheel. 0 = coarse not seen, 1 = coarse seen. |
| 02D0 | 1 | STEPS_REMAINING | Number of steps remaining to move the filter wheel a complete revolution when the coarse sensor was detected. Range [0,898 (hex)]. |
| 02D1 | 1 | FINAL_STEPS | A count of the steps to do when detecting the coarse sensor. Range [0,1257/1258]. |
| 02D2 | 1 | LAST_PHASED_USED | Last phase used when moving the filter wheel. Phases 1-4 are represented as 1111, 2222, 4444, 8888. |
| 02D3 | 1 | BAD_FW | Records errors encountered when moving the filter wheel. See note 2. |
| 02D4 | 1 | INIT_ICB | ICB settle loop. Number of loops remaining when the ICB status became OK. Counts down from DF37 (hex). |
| 02D5 | 1 | WPR_SAVE | Adascope variable. Not used. |
| 02D6 | 6 | cmdbuf | Adascope variable. Not used. |
| 02DC | 2 | ackbuf | Adascope variable. Not used. |
| 02DE | 16 | rstate | Register save area during interrupt handling. |
| 02EE | 3 | state | Interrupt linkage/service pointer storage. Machine state at time of interrupt. |
| 02F1 | 35 | ACCEL_TABLE | Constant. Table of values used for accelerating the filter wheel. |

Notes:

1. Progress flag.

   This is a bit pattern recording the safing procedure completed. The following table indicates which bit corresponds to each safing procedure. Initially each bit is set and is reset only after the procedure has been successfully performed. If the procedure is not appropriate for the type of boot in progress then the bit will remain set.

| Bit | 0-9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|
| Procedure | Not Used | High Voltage | Heaters | Phases | DPU | Second-aries | Filter Wheel |
| Warm | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Cold | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

   Therefore, after a cold start the value of this parameter should be 0022 (hex) and after a warm start it should be 0 if all the safing procedure were carried out successfully.

2. Filter Wheel Safing Error Counter.

   This comprises of three 4-bit nibbles in the least significant portion of the word (bits 4 to 15) as shown in the following table:

| Bits | 0-3 | 4-7 | 8-11 | 12-15 |
|---|---|---|---|---|
| Err or | Not Used | Bad Phase | Bad Sensor Read | Bad LED |

   If no errors occurred whilst safing the filter wheel then all the nibbles will be set to 0. Only one error will be recorded as the safing procedure is aborted after the first error is detected. The errors that can happen are all ICB command related and are shown in the following table:

| Error Code | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Meaning | Sync Error | Extensio n Error | Transmission Error | Time-out Error |

### 6.2.6 Routines

This section lists the most important routines used by the bootstrap, giving their address in code space.

| Name | Address | Description |
|---|---|---|
| KSTART | 01DF | Start of the Tartan supplied Adascope kernel code. |
| WARM_SAFE1 | 0314 | Code which is common to both Cold and Warm boots. Turns off the Heaters and Filter Wheel Phases and resets the DPU. |
| SAFE_FW | 032B | This routine performs the filter wheel safing. |
| ICB_CMD_SEND | 038F | Routine to send commands along the ICB bus. |
| UPDATE_PROGRESS | 03B7 | Routine called after each safing procedure to record the result of each particular safing procedure. |
| SAVE_STATS | 03BD | Saves the result of safing the Filter Wheel. |
| DELAY | 03C1 | Implements a delay in software. |
| CHECK_BPE | 03C9 | Routine to check whether the Blue Processing Electronics is on or off. Used to provide more information in determining whether the current Boot Type is warm or cold. |
| WATCHDOG_LOOP | 03D3 | Loop which is entered only when the Bootstrap has determined that it is running due to a Watchdog kick and after the instrument has been safed. |
| START_ICU | 03D9 | This routine puts the RBI into running mode and starts the Basic mode code. |
| SW_INDICATE | 03E7 | Writes to the 4 software indication bits available in the RBI configuration register. |
| WRF_START | 03F2 | Routine executed when the ICU has received an “ICU RESET” instruction to RBI. Relocated to address FFF8 in Code space by the bootstrap. |
| GOCMD | 016E | Part of the Adascope kernel which is called when the bootstrap has detected a warm start. Waits for an RBI “GO” command or an “ICU REST” instruction to RBI. |
| COPY_BOOTSTRAP<br>COPY_BOOTSTRAP_<br>AND_VECTORS | 0295 | Routine which copies the bootstrap code from the PROM into RAM. |
| COPY_BASIC | 0213 | Routine which copies the Basic mode code from the PROM into RAM. Executed unless following a “ICU REST WARM NO COPY” instruction to RBI. |
| COPY_BOOTSTRAP_<br>ONLY | 0292 | Routine which copies only the bootstrap code from PROM to RAM. Basic mode interrupt table is not copied. |

### 6.2.7 APPENDIX

**Private Communication 30-MAY-1996 15:47:02.24**

From: MSSL::JAT "Jason A Tandy"
To: ADV
CC: JAT
Subj: RBI Chip. Attn: P.Mercier.

Dear Philippe,

With respect to the RBI's Watchdog timer;

When this times out and resets the running bit in the Status register, how does the ICU software on rebooting go through the normal sequence of setting the Reset bit then the Running bit?
At present I find that I get a bad operation in the Configuration register.
Does the ICU software have to wait until the spacecraft has read the Status register and clears the Watchdog timeout bits?
Then the ICU can proceed.

Cheers, Jason.

**Private Communication 31-MAY-1996 08:08:53.92**

From: SMTP%"advtlse@dialup.francenet.fr"
To: jat@mssl.ucl.ac.uk (Jason A Tandy)
CC:
Subj: Re: RBI Chip. Attn: P.Mercier.

Dear Jason,

When a watchdog error is detected, then the error is flagged in the RBI status word by resetting the Running bit. However the RBI is still considered to be in the 'Running' state and not in the 'Init' state and then a microprocessor 'Reset ICU' and 'START ICU' instructions is considered as invalid (this explain why the bad operation bit is set in the Configuration register).

Note that a Watchdog time-out error indicates that the ICU SW has failed and then it is not able to issue these commands. In fact this is the central computer role to manage this error by issuing a 'Reset ICU' interrogation which will restart the ICU SW. This interrogation can be preceded by a 'Suspend ICU' interrogation and by 'Read Block' interrogations if the central computer wants to check the ICU memory before to restart the microprocessor.

Best regards.

P. Mercier

------------------------------------------------------------------------
- Philippe Mercier, ADV technologies -
- Parc Technologique du Canal, 16 Avenue de l' Europe -
- 31520 Ramonville Saint Agne, France -

- Tel : (33) 62 19 04 44 Fax : (33) 62 19 03 54 -
- E-mail advtlse@Dialup.FranceNet.fr -

## 6.3 Basic Code

Basic code is built from the following files:-

| ADA | | Assembler |
|---|---|---|
| Specifications | Bodies | |
| bcp4_ih.ads | | bcp4_ih.asm |
| | | bsio.asm |
| crc.ads | crc.adb | |
| debug.ads | debug.adb | |
| dempsu.ads | dempsu.adb | |
| | | emboot.asm |
| | | emsubs.com |
| hk.ads | hk.adb | |
| icb.ads | icb.adb | |
| icb_driver.ads | icb_driver.adb | |
| | icu.ada | |
| icu_mem_manager.ads | icu_mem_manager.adb | |
| importance.ads | | |
| mem_manager.ads | mem_manager.adb | |
| memloc.ads | | |
| modeman.ads | modeman.adb | |
| mutex.ads | mutex.adb | |
| nhk.ads | nhk.adb | |
| packet.ads | | |
| peek_poke.ads | | peek_poke.asm |
| rbi.ads | rbi.adb | |
| rbi_ih.ads | | rbi_ih.asm |
| reset.ads | | reset.asm |
| ssi_driver.ads | | ssi_driver.asm |
| ssi_ih.ads | | ssi_ih.asm |
| task_report.ads | task_report.adb | |
| taskman.ads | taskman.adb | |
| tc_q.ads | tc_q.adb | |
| tc_verify.ads | tc_verify.adb | |
| tcq.ads | tcq.adb | |
| time_man.ads | time_man.adb | |
| tm_man.ads | tm_man.adb | |
| tm_q.ads | tm_q.adb | |
| tmpsu.ads | tmpsu.adb | |
| tmq.ads | tmq.adb | |
| types.ads | | |
| | | USERDEFS.ASM |

The following pages contain 'Structured English' extracted from comments in the file. They should be studied in conjunction with the code listings as they have additional comments regarding implementation details but are omitted in this document for clarity.

- The comments extracted from the specification files (*.ads) describe '**what'** a given package does.
- The comments extracted from the associated body files (*.ads or *.asm) describe '**how'** a given package performs the operations defined by the specification.

In addition, the file `icu.xtof` can be supplied. It may be used in conjunction with the TARTAN utility `adaref1750a` to extract the dependencies, list of calls and inverse calls and cross reference information..

To extract the call graph (of 'callers').

```
adaref1750a –input icu.xtof –call_graph
```

To extract the call graph (of 'called by').

```
adaref1750a –input icu.xtof –call_graph –reverse
```

To extract the call graph (of 'callers') from one package.

```
adaref1750a –input icu.xtof –call_graph –from package_name
```

To extract a list of dependent relationships.

```
adaref1750a –input icu.xtof –dependency_graph
```

To extract a list of dependent relationships from one package.

```
adaref1750a –input icu.xtof –dependency_graph –from package_name
```

To extract a alphabetical list of user defined entities, containing source location of declaration, source location of where it is set and used.

```
adaref1750a –input icu.xtof –xref
```

To extract a alphabetical list of user defined entities, containing source location of declaration, source location of where it is set and used for one package.

```
adaref1750a –input icu.xtof –xref –about package_name
```

## 6.3.1 Main Program

*6.3.1.1 icu.ada*

**Extracted from file "icu.ada"**

```
 Function
 ========

 This procedure is the 'main' program for the basic code of the ICU. It

 1) Initialises the ICU then...
 2) Routes all valid received telecommand packets as appropriate

procedure ICU is


   Initializations
   ===============


   Initialise the SSI interface controlling software.

   Initialise RBI related matters
   (including the communications area and TC and TM ready bits)

    Start the RBI Watchdog.

    Ensure that telemetry queues are initialised

    Ensure the telecommand queues are initialised (after which we can
    receive telecommands

    Send the Bootstrap Status Block

    1st Determine whether its an event (boot OK) or exception (boot not OK)

    then send the block

    Now turn on both main heaters, in order to compensate for lack of
    heat input because secondaries are not on during basic mode.

    Now start the Housekeeping task

    Now begin the endless control loop

       Wait for a valid telecommand packet

       When a valid packet is obtained, route it to the appropriate package
       on the basis of the packet type

          For a Task Management Packet

             send it to the Task Manager package TASKMAN

          For a Memory Maintenance Packet

             call the memory manager package MEM_MANAGER

          For a Telemetry Management Packet

             Call the telemetry manager package TM_MAN.

          For a Time Management Packet

             Call the Time Manager package TIME_MAN

          For a test packet

             do nothing

          For all other packet types

             do nothing

       end of selection by packet type

       If nothing has indicated that the packet was bad
```

```
      Place a Successful Acceptance Telemetry Packet in the
      telemetry queue.

      Increment the good packet count (modulo 65536) for HK purposes.

      Otherwise, increment the bad packet count (modulo 65536)
      for HK purposes

End the controlling loop
```

### 6.3.2 Packages

#### *6.3.2.1 bcp4_ih.ads*

**Extracted from file "bcp4_ih.ads"**

```
Function
========

This file merely contains the specification for the XMM-OM bcp4 interrupt
handler.  It specifies that the body of bcp4_ih is written in assembler
and therefore directs the linker to link it as foreign.
The interrupt handler had to be written in assembler for speed so as not to
block other interrupts for too long.
```

#### *6.3.2.2 bcp4.ih.asm*

**File is bcp4_ih.asm**

```
         Fetch the interrupt counter
            Check for impending overflow
            If it's OK, increment it
            otherwise avoid overflow
    Check BCP flag and if it is not 1, we don't have to bother so jump to end
    "Freeze" the current time by writing appropriate instruction
    to config register.
    Read bits 0-15
    Read bits 16-31
    Read remaining bits 32-42 (result in high order bits)
    Set the BCP flag to 2 to show we've got a time
            Recover registers
            Turn on interrupts
            Back from whence we came
```

#### *6.3.2.3 bsio.asm*

**File is bsio.asm**

```
Name
    INITLINK

Initialize the communications link

Parameters
    None

Notes
      This routine is called on startup. R14 is the link register.
      All other registers may be trashed.
      In the ROM version, this routine is called after the kernel has
      been copied to RAM, but before the startup ROM is shut off.
      This code may either execute from ROM, or disable the startup
      ROM if it needs to read RAM.

      NO LONGER USED


Name
      QUIET

Presuming a transmission error, wait for quiet on input link

Parameters
      None
Notes
      R14 is the link register.  R0, R1 and  R3 can be trashed.
      We 'read' and discard characters until there had been no more input
      for 500ms.
      NO LONGER USED


Name
      ENABLE_MONITOR

Enable monitoring of the link before going off to the user's program

Parameters
      None
Notes
      R14 is the link register.  All other registers are trashable.
      Usually, we enable UART receiver error or data interrupts.
      Thus, if the host tries to send us a message while we are in
      the user's program we will get back to the kernel (we hope).

(?)   In the SBC50 we left the interrupt on.  We just clear the pending
(?)   bit, if set.

      NO LONGER USED


Name
      READLINK

      Read bytes from the communications link

Parameters
      r12    Destination address
      r0     Byte count (must be even)
      r9     Address State

Returns
      checksum in r0
Notes
      R14 is the link register. Destroys r1,r2,r3
      but r2 counts down to 0 for cmdinterp to check.
      r12 used later too



READLINK       EQU                                                  $


RDRDY
```

```
        Set the software indication bits to 2

RD_POLL

        Read the RBI configuration register
        If IT1 (interrupt) pin has been asserted
        then branch to STARTTEST
        else branch to RD_POLL

STARTTEST
                ; Don't forget ICU Resetw command

        Read instruction to RBI register
        If there has been a start ICU command
        then jump to START_ICU
        If it is not a reset command then branch to RD_POLL
        Reset command so jump to 16#FFF8# in page 2


 Name
        Writelink

        Write bytes to the communications link

 Parameters
        r12     Destination address
        r0      Byte count (must be even)

 Notes
        R14 is the link register. Destroys r1,r2,r3,R13
        mov ra,r12 ; ra=move to rbi addr; r12=move from, r(a+1)=number to move



WRITELINK      EQU                                                          $
WRRDY

        Not used
```

#### 6.3.2.4 crc.ads

Extracted from file "crc.ads"

```
    Function
    ========

    This file contains the specification for the CRC package.
    This contains the CRC algorithms for XMM which
    are based on the algorithm described in ESA technical note PX-TN-00540

--------------
package CRC is
--------------


      This function returns the unsigned 16 bit integer checksum of the
      first NUMBER locations in unsigned byte array DATA.

   function CHECK_TC(TC : PACKET.TC_TYPE) return UINT16;


      This function calculates the checksum of telecommand packet TC,
      using the packet length stored within the packet to determine its
      length. Returns value of zero if as expected, otherwise returns
      value of checksum found, NOT including the 2 byte checksum
      field at the end of the packet.
      It thus checks whether that packet TC contained a valid CRC.

   function CALC_TM(TM : PACKET.TM_TYPE) return UINT16;


      This function calculates the value to be inserted into
      the checksum field of packet TM, using the packet length stored
      within the packet to determine the length of the data to be checksumed
      (i.e. NOT including the checksum field at the end of the packet).

   function CALC_MEM(CURRENT_CRC : UINT16;
                     MEM : UINT16_ARRAY;
                     NO_WORDS : INTEGER) return UINT16;


      This function is used to calculate a checksum for a large block
      of data on the assumption that not all the data will be available
      at once. Therefore, it uses the CURRENT_CRC value returned by a prior
      call as input to the current call and then calculates the CRC of the
      NO_WORDS 16-bit words of data contained in MEM. The result is the CRC
      for all blocks of data supplied (NOTE: the sequence is restarted by
      supplying a value of all binary ones for CURRENT_CRC).
```

#### *6.3.2.5 crc.adb*

Extracted from file "crc.adb"

```
 Function
 ========

 This file contains the body for the CRC package.
 This contains the CRC algorithms for XMM which
 are based on the algorithm described in ESA technical note PX-TN-00540

-------------------
package body CRC is
   -------------------


   function CLC(SYNDROME : UINT16; DATA : UBYTE_ARRAY;  NUMBER : UINT16 )
                return UINT16 is


     This function returns the unsigned 16 bit integer checksum of the
     first NUMBER locations in unsigned byte array DATA. An initial value
     of the currently 'running' checksum is contained in SYNDROME.
     It is a function internal to this package.

     The following test data was used (taken from the reference above).

          DATA                 CRC
          ++++                 +++
     00 00                    1D 0F
     00 00 00                 CC 9C
     AB CD EF 01              04 A2
     14 56 F8 9A 00 01        7F D5

   First define the lookup table for efficient calculation (equivalent of
   routine InitLtbl in above reference.

      loop over NUMBER data points

        Calc RHS term by

        1) Shift right the input checksum by 8.

        2) Exclusive Or result with current datum.

        3) Mask off the 8 least significant bits of the result.

        4) Use result to index into table of pre-calculated coefficients.

        calc LHS term by

        1) Shift left the input checksum by 8.

        2) Mask off the 8 most significant bits of the result.

        Calculate checksum by Exclusive Oring the two terms.

      Return final value of the checksum.

   function CALC(DATA : UBYTE_ARRAY;  NUMBER : UINT16 ) return UINT16 is


      Call the CLC routine with the initial CRC set to all binary 1's.

   function CHECK_TC(TC : PACKET.TC_TYPE) return UINT16 is


      This function calculates the checksum of a whole packet,
      using the packet length stored within the packet to determine its
      length. Returns value of zero if OK, otherwise returns
      value of checksum found, NOT including the 2 byte checksum
      field at the end of the packet.
      It thus checks whether that packet contained a valid CRC.

      Call routine CALC (using the whole packet as data and deriving
      its length from internal length information) to check that the result
      (i.e. the checksum of whole packet) is zero

        if it is, return zero

        Otherwise
```

```
     Return checksum found (not including the CRC field).

function CALC_TM(TM : PACKET.TM_TYPE) return UINT16 is


   This function calculates the value to be inserted into
   the checksum field of packet TM, using the packet length stored
   within the packet to determine the length of the data to be checksumed
   (i.e. NOT including the checksum field at the end of the packet).

   Calculate the appropriate length to be used from the length
   field in the packet, then use routine CALC to calculate the
   checksum of packet TM and return the value.

function CALC_MEM(CURRENT_CRC : UINT16;
                  MEM : UINT16_ARRAY;
                  NO_WORDS : INTEGER) return UINT16 is


   This function is used to calculate a checksum for a large block
   of data on the assumption that not all the data will be available
   at once. Therefore, it uses the CRC value returned by a prior
   call as input to the next one.

    Loop over the block of data, 1 16 bit word at a time.

      Call function CLC to calculate the 'running' CRC for just 1 word.

    Return the resulting CRC.
```

#### *6.3.2.6 debug.ads*

Extracted from file "debug.ads"

```
    Function
    ========

    This file contains the specification and body for the package DEBUG.
    As its name implies, it contains a collection of routines useful
    for debugging.
    Both procedures write a meaningful number to fixed location in memory
    which can be read later (e.g., after a crash) to help understand what
    went wrong.

    Dependencies
    ============

with TYPES; use TYPES;
with SYSTEM;
with MEMLOC;


----------------
package DEBUG is
   ----------------


   ---------------------------------
   procedure PROGRESS(ITEM : UINT16);
   ---------------------------------


      Where ITEM is the progress number to write to memory
      This procedure writes the number "ITEM" to a fixed location in memory
      and is used to keep a record of how far the running code has progressed.
      When this memory location is read later, after a crash, it will provide
      good idea as to what was running as the code crashed.

   ---------------------------------
   procedure EXCEPTION_REPORT(ITEM : UINT16);
   ---------------------------------


      Where ITEM is the exception number to write to memory
      When the running code produces an Ada exception, the Ada exception
      handler should call this procedure which will write the exception
      number to a special known location in memory that can be read afterwards
      to help understand why the code crashed.

      Define some constants for the progress numbers.
      In this way, the high order bits of the code numbers used indicate the
      package involved.
```

#### *6.3.2.7 debug.adb*

Extracted from file "debug.adb"

```
 Function
 ========

 This file contains the body for the package DEBUG.
 As its name implies, it contains a collection of routines useful
 for debugging.

package body DEBUG is


   procedure PROGRESS(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory

       If we haven't had an Ada exception

          Write ITEM to the FIRST_PROGRESS standard memory location
          ITEM identifies which part of the code is running: the package and
          a location in that package
          After an Ada exception the value stored at this address
          will not change

       Write ITEM to the LAST_PROGRESS standard memory location
       This will continue to update after an Ada exception

   procedure EXCEPTION_REPORT(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory

       If this is the first exception trapped

          Write ITEM to the fixed memory location reserved to store the
          first exception.  This will not be overwritten.
          ITEM identifies in which part of the code the exception occured:
          the package and which exception was handled

       Then write ITEM to the fixed memory location reserved to store the
       last exception.  This is overwritten at each exception.
```

#### *6.3.2.8 dempsu.ads*

Extracted from file "dempsu.ads"

```
    Function
    ========

    This file contains the specification for the DEMPSU package
    It provides routines to control the Digital Electronics Module
    Power Supply Unit.

-----------------
package DEMPSU is
-----------------


  procedure DPU_RESET;


      Resets the DPU after a 'latch-up' or turns it on again if it is
      powered down.
```

#### 6.3.2.9 dempsu.adb

```
Extracted from file "dempsu.adb"


 Function
 ========

 This file contains the body for package DEMPSU
 It provides routines to control the Digital Electronics Module
 Power Supply Unit.

----------------------
package body DEMPSU is
   ----------------------


    Define the addresses used

    The DEMPSU reset register := DPU_RESET_REGISTER

    Define the procedure/functions to read / write to registers

   procedure DPU_RESET is


      To reset/turn on the DPU, write a "don't care" bit
      pattern to the DPU Reset Register of the DEMPSU control card.
```

#### *6.3.2.10 hk.ads*

Extracted from file "hk.ads"

```
    Function
    ========

    This file defines the specification for the HK package. The package
    acquires and sends the Housekeeping Packets (HK), the contents of
    which are defined in the XMM-OM Telecommand and
    Telemetry Specification document, XMM-OM/MSSL/ML/0010

-------------
package HK is
 -------------


   procedure ON;


       This procedure enables the acquisition of the HK packet type

   procedure OFF;


       This procedure disables the acquisition of the HK.
```

#### *6.3.2.11 hk.adb*

Extracted from file "hk.adb"

```
 Function
 ========

 This file defines the body for the HK package. The package
 acquires and sends the Housekeeping Packets (HK), the contents of
 which are defined in the XMM-OM Telecommand and
 Telemetry Specification document, XMM-OM/MSSL/ML/0010

------------------
package body HK is
------------------


    Create an array of flags to hold the individual 'HK packet
    is enabled' status

   task PROCESS is
      pragma PRIORITY(IMPORTANCE.HK_PROCESS);
      entry ON;
      entry OFF;
   end PROCESS;


   The above is the specification for the internal task that performs the HK
   acquisition

   Entry ON starts the task.
   Entry OFF stops the task
     and returns whether or not it was already stopped.

   task body PROCESS is


       Create an instance of an HK packet

      Set up initial time interval

      Commence infinite loop

       Await for either:

         1) A request to start HK acquisition (already on by default)

               If ON request comes in

               Initiliase the next time for HK to be now

          2) A request to stop HK acquisition

            If OFF request comes in

               then disable acquisition

         3) otherwise, provided HK is enabled (the default)

                wait until it's time to collect the next block of HK

                unless the time is too negative

                   Decide which HK section to acquire

                   and branch accordingly

                      If its the Detector section

                         Take no action

                      If it's the TMPSU

                         Get Heater status

                         Get Sensor current info

                         Get Secondary Voltages

                         Get TMPSU Secondary Currents
```

```
                If it's the ICB section

                   Get Status of ICB

                If it's the SSI section

                   Get SSI I/F error count

                If it's the RBI section.

                   Get RBI Status and Configuration Registers

                   DEMPSU Voltages

                If it's the miscellaneous section

                   Get ICB Error Count

                   Get TC Good Packet Counter

                   Get TC Bad Packet Counter

                   Get OM State

                   Get ICU State

                   Get Which chain (i.e Prime or Redundant)

                   Get S/W Version

                If it's the DPU section.

                   Get DPU Info

                   Correct for DPU ROM bug (NCR 89)

                If it's the section where we send out the packet.

                   then set the HK Packet SID field accordingly

                   Get the current time and place in packet

                   Indicate CRC present

                   Calculate and set the packet length field

                   Provided at least one type of HK SID is enabled

                      Send packet to telemetry queue

             Set up for next HK section

          Check whether current SID has changed

          Calculate the next HK sample time
          (derived from the time determined at start and the SID)

          Subtract it from the current time and delay the
          code by the result, thus ensuring an average time interval

    end of infinite loop

procedure OFF is


    Disable the HK acquisition program

procedure ON is


    Ensure HK program is running
```

#### *6.3.2.12 icb.ads*

Extracted from file "icb.ads"

```
    Function
    ========

    This file contains the specification for the ICB package. The package
    controls access to lower-level routines that interface directly with
    the Instrument Control Bus (ICB). The ICB is implemented using the
    MACSbus protocol.

--------------
package ICB is
--------------


   task GUARDED is

      pragma PRIORITY(IMPORTANCE.ICB_GUARDED);

      entry PUT(DEST      : DEST_ADDRESS_TYPE;  -- data to one sub-address
                SUBADR    : SUB_ADDRESS_TYPE;
                DATUM     : UINT16;
                OK        : out BOOLEAN);

      entry GET(DEST      : DEST_ADDRESS_TYPE;
                SUBADR    : SUB_ADDRESS_TYPE;
                DATUM     : out UINT16;
                OK        : out BOOLEAN);

      entry RESET;

   end GUARDED;


       Provides one-at-a-time controlled access to the PUT, GET and RESET
       functions for the ICB.

       PUT
         Writes DATUM to sub-address SUBADR at MACSbus destination DEST.
         Returns OK = TRUE if no errors occur.

       GET
         Reads DATUM from sub-address SUBADR at MACSbus destination DEST.
         Returns OK = TRUE if no errors occur.

       RESET
         Resets the ICB MACSbus interface.

   function REPORT(TID : UBYTE;
                   FID : UBYTE) return BOOLEAN;


       The function implements the "Read ICB Address Directly" command
       as described in section 2.2.5 of the Telecommand and Telemetry
       Specification, XMM-OM/MSSL/ML/0010.

       Specifically, it constructs a Task Parameter Report [TM(5,4)] containing
       the datum read back from subaddress FID at destination TID-40(hex), as
       documented in section 3.5 of the above document.

       In this release, it always returns TRUE.

   function STATUS return UBYTE renames ICB_DRIVER.HK_STATUS;


       For convenience, renames a low-level routine which returns
       the ICB interface status word - see package ICB_DRIVER for
       more details.

   function ERROR_COUNT return UBYTE renames  ICB_DRIVER.ERROR_COUNT;


       Returns the ICB error count (modulo 256) since the ICU was started.
```

#### *6.3.2.13 icb.adb*

Extracted from file "icb.adb"

```
 Function
 ========

 This file contains the body for the ICB package. The package
 controls access to lower-level routines that interface directly with
 the Instrument Control Bus (ICB). The ICB is implemented using the
 MACSbus protocol.

-------------------
package body ICB is
   -------------------


   task body GUARDED is


       Reset Interface

       Commence Infinite Loop

          Await a call on one of the following:

             If a call to RESET is made

                Call the ICB driver RESET procedure from ICB_DRIVER.

             If a call is made to the PUT procedure in ICB_DRIVER.

                Send the data to the put ICB driver

             If a call is made to the GET entry

                Obtain a value via the GET procedure from ICB_DRIVER.

          End of infinite loop

   function REPORT(TID          : UBYTE;
                   FID          : UBYTE) return BOOLEAN is


       Get the datum at the address and sub-address corresponing
       with the supplied TID and FID.

       Supply the datum to the TASK_REPORT package to construct
       and send the aappropriate Report Task Parameters Packet.

       Always return success.
```

#### *6.3.2.14 icb_driver.ads*

Extracted from file "icb_driver.ads"

```
    Function
    ========

    This file contains the specification for the ICB_DRIVER package.
    The package provides the lower-level routines that interface directly
    with the Instrument Control Bus (ICB). The ICB is implemented using the
    MACSbus protocol.

---------------------
package ICB_DRIVER is
   ---------------------


   procedure PUT(DEST    : DEST_ADDRESS_TYPE;
                 SUBADR  : SUBADR_ADDRESS_TYPE;
                 DATUM   : UINT16;
                 OK      : out BOOLEAN);


       This procedure write the datum DATUM to sub-address SUBADR at
       MACSbus destination DEST. OK is set to TRUE if no errors occur.

   procedure GET(DEST    : DEST_ADDRESS_TYPE;
                 SUBADR  : SUBADR_ADDRESS_TYPE;
                 DATUM   : out UINT16;
                 OK      : out BOOLEAN);


       This procedure request the datum DATUM from sub-address SUBADR at
       MACSbus destination DEST. OK is set to TRUE if no errors occur.

   procedure RESET;


       This procedure resets the MACSbus interface.

   function HK_STATUS return UBYTE;


       Returns ICB status
       BUT only for the last occurring error.

   function ERROR_COUNT return UBYTE;


       This returns the (modulo 256) error count of MACSbus errors since
       the ICU code started running.
```

#### *6.3.2.15 icb_driver.adb*

```
Extracted from file "icb_driver.adb"


 Function
 ========

 This file contains the body for the ICB_DRIVER package.
 The package provides the lower-level routines that interface directly
 with the Instrument Control Bus (ICB). The ICB is implemented using the
 MACSbus protocol.

--------------------------
package body ICB_DRIVER is
--------------------------


    Define the structure of the status register

    --------------------------------------------------------
    | msb |     |     |     |     |     |     | lsb |
    --------------------------------------------------------
    |  8  |  9  | 10  | 11  | 12  | 13  | 14  | 15  |
    --------------------------------------------------------
    |     DEAD BITS   |     | TX  | EXT | SYNC|  END |
    |                 |     | ERR | ERR |  ERR| COMM |
    --------------------------------------------------------

    Note: the structure of the ICB command register is:

    --------------------------------
    | MSB                      LSB|
    --------------------------------
    | ext |  dest   | subadr  |inst |
    |                               |
    | | | | | | | | | | | | | | | | |
    --------------------------------

   function GET_STATUS return ICB_STATUS_TYPE is


      Read the ICB MACSbus status register port.

      Extract and return the status word

   function HK_STATUS return UBYTE is


      Return the last noted **error** status word.

   procedure PUT(DEST    : DEST_ADDRESS_TYPE;
                 SUBADR  : SUBADR_ADDRESS_TYPE;
                 DATUM   : UINT16;
                 OK      : out BOOLEAN) is


      Construct command word to be written to command register
      based on supplied DEST and SUBADR
      (Note, Instr = RD = 010 binary, Ext = 101 binary)

       Write Datum to datum register port

       Write command word to command register (thus initiating transfer)

       Wait for completion of command (END COMM bit set),
       an error (i.e. TX ERR, EXT ERR or SYNC ERR bit set) or a timout, and
       remember the resulting status.

       Flag an error if error bitset , a timout or all 'dead bits' set.
       Otherwise, assume OK.

       if no error

         Do nothing.

       Otherwise

         Hand status, command word and datum over to be
         processed by the Analyse Errors procedure.

       Finally, ensure status register always reset by
```

```
    calling procedure RESET.

procedure GET(DEST    : DEST_ADDRESS_TYPE;
              SUBADR  : SUBADR_ADDRESS_TYPE;
              DATUM   : out UINT16;
              OK      : out BOOLEAN) is


   Construct word to be written to command register
   based on supplied DEST and SUBADR
   (Note, Instr = TI = 100 binary, Ext = 101 binary)

    Write command word to command register
    (which initiates transfer).

    Wait for completion of command (END COMM bit set),
    an error (i.e. TX ERR, EXT ERR or SYNC ERR bit set) or a timout, and
    remember the resulting status.

    Set OK as 'false' if error or timout or all dead bits set
    Otherwise set 'true'

    Get datum (this will be bad data if there was an error )

    If no error

        Do nothing.

    Otherwise

      Hand status, command word and datum over to be
      processed by the Analyse Errors procedure.

    Finally, ensure status register always reset by
    calling procedure RESET.

procedure RESET is


    To reset the ICB interface, write a "don't care" bit
    pattern to the Status Register port.

    Note new status.

procedure ANALYSE_ERRORS(COMMAND_WORD : UINT16;
                         DATUM : UINT16;
                         STATUS: ICB_STATUS_TYPE) is


    Remember this error status.

    Increment the error count (modulo 256)

  Construct and send the appropriate 'MACSbus Error' Exception Report.

function ERROR_COUNT return UBYTE is


   Return the (modulo 256) error count.
```

#### *6.3.2.16 icu_mem_manager.ads*

Extracted from file "icu_mem_manager.ads"

```
        function load_memory loads memory corresponding to the MID

        where MID is the MID
        where START_ADDRESS is the start address of the load
        where DATA is the data to load as an array of unsigned 16 bit words
        where LENGTH is the length of the data in words
        where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
        returns a boolean: true on success and false on failure

        function dump_memory dumps memory corresponding to the MID

        where MID is the MID
        where ADDRESS is the address of the dump request
        where LENGTH is the length of the requested memory dump in words
        where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
        returns a boolean: true on success and false on failure

        function calculate_memory_checksum calculates the checksum of the memory region
corresponding to the MID

        where MID is the MID
        where ADDRESS is the address of the crc request
        where LENGTH is the length of the requested block of memory to crc in words
        where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
        returns a boolean: true on success and false on failure
```

#### *6.3.2.17 icu_mem_manager.adb*

Extracted from file "icu_mem_manager.adb"

```
 Dependencies
 ============

with TYPES; use TYPES;
with UNCHECKED_CONVERSION;
with ARTCLIENT;
with PACKET;
with TC_VERIFY;
with TMQ;
with PEEK_POKE;
with CRC;
with TIME_MAN;
with SYSTEM;
with NHK;


-------------------------------
package body ICU_MEM_MANAGER is
   -------------------------------


   task MEMORY_DUMP is


   procedure SEND_PACKET(SUB_TYPE: PACKET.TELEMETRY_SUBTYPE; ADDRESS: LONG_INTEGER; DATA :
UINT16_ARRAY; LENGTH : UINT16; MID: UINT16) is
      CRC_LENGTH: UINT16;
      DUMP_PACKET: PACKET.TM_TYPE(PACKET.MEMORY_MAINTENANCE_REPORTS, SUB_TYPE);


       Flag CRC as present

       Check if CRC is present

       If subtype is for a memory_dump

           Write the address into the packet

          Write the packet_length into the packet

          Write the data into the packet

          If subtype is for a memory_checksum_report

           Write the address into the packet

          Write the packet_length into the packet

          Write the memory_length into the packet

       Send the packet

   procedure READ_BLOCK(MID: UINT16; ADDRESS: LONG_INTEGER; LENGTH: INTEGER; DATA: in out
UINT16_ARRAY; SEQUENCE_COUNT_AND_SOURCE: UINT16) is


       returns array 0 .. PACKET.MAX_TM_MEM_PARAMS_M1

       Check the MID

             When the MID is 0: icu operand/data space
             For each word of data to be read

                Calculate the address state

                Enter critical section

                Read from the address

                Leave critical section

             When the MID is 1: icu instr space
             For each word of data

                Calculate the address_state

                Enter critical section
```

```
            Read from the address

            Leave critical section

         When the MID is wrong
         Send unsuccessful acceptance packet

   task body MEMORY_DUMP is


      begin an infinite loop

         if a call to start is made

            Finish when there's nothing left

            If there's more than a packet left

               Read the memory

               Send the data in a packet

               Recalculate the no of words left

               If there's less than or just one packet left
               Read the memory

               Send the data in a packet

   function LOAD_MEMORY(MID: UINT16; START_ADDRESS: LONG_INTEGER; DATA: UINT16_ARRAY; LENGTH:
UINT16; SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


         When the MID is 0: icu operand/data space
         For each word to be loaded

            Calculate address state and address offset

            Protect from address state change by entering critical section

            Write the value to memory

            Leave critical section

         When the MID is 1: icu instruction space
         For each word to be loaded

            Calculate address state and address offset

            Protect from address state change by entering critical section

            Write the value to memory

            Leave critical section

         Otherwise the MID must be wrong
         put params in array

         Send unsiccessful acceptance (illegal mid) packet

   function DUMP_MEMORY(MID: UINT16; ADDRESS: LONG_INTEGER; LENGTH: UINT16;
SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


      Remember the dump parameters

      Try to ask for dump

         for 0.5 second

         if can't dump, return false so that an unsuccessful execution can be sent

   function CALCULATE_MEMORY_CHECKSUM(MID: UINT16;
                                      ADDRESS: LONG_INTEGER;
                                      LENGTH: UINT16;
                                      SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


      Set crc syndrome to ffff to start with

      loop
```

```
    until there's nothing left to crc

    If there's more than or just one packet's worth left

       Read a block of memory

       crc it

       recalculate length remaining

       If there's less than a packet's worth left
       Read a block of memory

       crc it

       finish

 Send a memory checksum report with the checksum just calculated
```

#### 6.3.2.18 importance.ads

Extracted from file "importance.ads"

```
    Function
    ========

    This package defines the priority of tasks

    The range of priorities is 10..200
    The default is SYSTEM.DEFAULT_PRIORITY := 10;

    Priorities are allocated in bands as follows:-

    H/W Simulators (for debugging)               191 -> 200
    CPU Watchdog reset                           190
    S/W Watchdogs                                171 -> 189
    "Guard" Tasks to control access to resources 151 -> 170
    Task initiated by interrupts                 141 -> 150
    "Semaphore" Tasks                            131 -> 140
    "Monitor Tasks" (eg. DPU, TM)                111 -> 130
    "Working Tasks" e.g. HK, Science, Blue       11  -> 110
    "Idle" Task                                  10

---------------------
package IMPORTANCE is
   ---------------------


       Priority Definitions
       ====================

       CPU Watchdog Reset
       ------------------

   CPU_RESET             : constant SYSTEM.PRIORITY := 190;


       Software Watchdogs
       ------------------

       DPU Heartbeat Watchdog Task

   DPU_HEARTBEAT         : constant SYSTEM.PRIORITY := 171;


       "Guard Tasks" to control access to resources
       -------------

         Priority of task to control access to SSI i/face

   SSI_GUARDED           : constant SYSTEM.PRIORITY := 151;


         Priority of task to control access to ICB i/face

   ICB_GUARDED           : constant SYSTEM.PRIORITY := 152;


         Priority of task to control access to telemetry queue

   TMQ_GUARDED           : constant SYSTEM.PRIORITY := 153;


         Priority of task to control access to HK record (NOT USED)

   HK_ACCESS             : constant SYSTEM.PRIORITY := 154;


         Priority of task to guard running/not running status flag for
         HK acquire (NOT USED)

   HK_RUNNING_GUARD      : constant SYSTEM.PRIORITY := 155;


       High Priority Interrupt Initiated Tasks
       ---------------------------------------

         Priority of BCP4 interrupt task

   BCP4_INTERRUPT        : constant SYSTEM.PRIORITY := 140;
```

```
    "Semaphore" Tasks
    -----------------

      Priority of DPU Event semaphore task

EVENT_ACTION          : constant SYSTEM.PRIORITY := 131;

      Priority of Mutual exclusion semaphore task type

MUTEX_SEMAPHORE       : constant SYSTEM.PRIORITY := 132;

      Timer A Resource

TIMER_A               : constant SYSTEM.PRIORITY := 133;

    "Monitor Tasks" (eg. DPU, TC)
    ---------------

      Priority of Task to monitor DPU data for events

DPU_DATA_MANAGER      : constant SYSTEM.PRIORITY := 112;

      Priority of Task to monitor Telecommand queue

TCPROC                : constant SYSTEM.PRIORITY := 111;

    "Working Tasks" (e.g. HK, Science, Blue)
    ---------------

      Priority of task that collects and send HK data

HK_PROCESS            : constant SYSTEM.PRIORITY := 92;

    Load Blue Centroid Table (NOT USED IN BASIC)

LOAD_CENTROID_TABLE   : constant SYSTEM.PRIORITY := 93;

    Load Blue Window Table (NOT USED IN BASIC)

LOAD_WINDOW_TABLE     : constant SYSTEM.PRIORITY := 94;

      Priority of task to perform Thermal Control (NOT USED IN BASIC)

THERMAL_CONTROL       : constant SYSTEM.PRIORITY := 95;

      Priority of task that fetches DPU science data (NOT USED).

FETCH_DPU_DATA        : constant SYSTEM.PRIORITY := 96;

      Priority of task that fetches other DPU data
      (e.g. priority data ) - NOT USED AS NOT IMPLEMENTED

DPU_OTHER_DATA_MANAGER: constant SYSTEM.PRIORITY := 97;

    IDLE Task (NOT USED)
    ---------

IDLE                  : constant SYSTEM.PRIORITY := 10;
```

#### *6.3.2.19 mem_manager.ads*

Extracted from file "mem_manager.ads"

```
--------------------------------------------------------------------
function REQUEST(MEM_MANAGER_PACKET: PACKET.TC_TYPE) return BOOLEAN;
--------------------------------------------------------------------


    Where MEM_MANAGER_PACKET is a memory management packet
    Returns BOOLEAN true success or false on failure
    This merely forwards packets onto the ICU_MEM_MANAGER
```

#### *6.3.2.20 mem_manager.adb*

Extracted from file "mem_manager.adb"

```
 Function
 ========

 This file contains the body for package mem_manager.
 It calls icu_mem_manager or dpu_mem_manager to load/dump/check memory.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/SP/0061

 Dependencies
 ============

with UNCHECKED_CONVERSION;

with PACKET;
with ICU_MEM_MANAGER;
with TMQ;
with TC_VERIFY;
with DEBUG;


---------------------------
package body MEM_MANAGER is
   ---------------------------


   function REQUEST(MEM_MANAGER_PACKET: PACKET.TC_TYPE) return BOOLEAN is


       Find length of CRC (is it there or not)

       Calculate length of data is packet

       Convert length from bytes to words

          Check memory management packet subtype - load/dump/crc

             Check address is valid

                If not, send an unsuccessful acceptance packet

             Check the MID

                When the MID is for the ICU

                   Call LOAD_MEMORY in ICU_MEM_MANAGER

                   Otherwise send an unsuccessful acceptance packet

                Return FALSE if something went wrong

             When it's a dump memory command (subtype 2)

             Check the MID

                When the MID is for the ICU (0, 1)

                   Call DUMP_MEMORY in ICU_MEM_MANAGER

                   Otherwise send an unsuccessful acceptance packet

                if we had trouble, send an unsuccessful execution packet

             When it's a memory crc (subtype 3)
             Check the MID

                If the MID is for the ICU (0, 1)

                   Call CALCULATE_MEMORY_CHECKSUM in ICU_MEM_MANAGER

                   Otherwise send an unsuccessful acceptance packet

             Otherwise we have a wrong subtype for MEM_MANAGEMENT
             So send an unsuccessful acceptance
```

#### *6.3.2.21 memloc.ads*

Extracted from file "memloc.ads"

```
    Function
    ========

    This file contains the specification only package MEMLOC.
    This package defines any fixed memory locations.

-----------------
package MEMLOC is
-----------------


       Define the location of the ADASCOPE version ID we are running

       Define the size of the telemetry queues

       Define RBI Communication Area Location

       Define the location TC_LOC of the telecommand queue area

       Define the location TM_LOC of the telemetry queue area

       Define other tc/tm special addresses (e.g. queue pointers)

       Define BCP4/RBI interrupt processing save areas
       (these are fixed to assist assembler
       and ADA routines to communicate with each other).

       define RBI special addresses

       Define Time Control Flags locations

       Define the Bootstrap Parameter Area

       Define SSI special address
```

#### *6.3.2.22 modeman.ads*

Extracted from file "modeman.ads"

```
------------------
package MODEMAN is
------------------

function TO_MODE(MODE : UINT16; PARAM : UINT16; SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN;

    Sets the current mode of the ICU.

function MODE return UINT16;

    Returns the current mode of the ICU.
```

#### *6.3.2.23 modeman.adb*

Extracted from file "modeman.adb"

```
with RESET;
with DEBUG;
with TC_VERIFY;
with PACKET;


-----------------------
package body MODEMAN is
-----------------------


function TO_MODE(MODE : UINT16; PARAM : UINT16; SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN is

       If MODE is full safe then

          Accept telecommand

          Wait one second for acknowledgement to be sent

          Set current mode to new mode

       Else

          Send unsuccessful command acceptance

function MODE return UINT16 is

       Return the current mode
```

#### *6.3.2.24 mutex.ads*

Extracted from file "mutex.ads"

```
    Function
    ========

    This file contains the specification for the MUTEX package. This
    provides a mutual exclusion semaphore emulation;

----------------
package MUTEX is
----------------

   task type SEMAPHORE is

      entry SEIZE;

        This entry point acquires the resource

      entry RELEASE;

        This entry point releases the resource

   end SEMAPHORE;

end MUTEX;
```

#### *6.3.2.25 mutex.adb*

Extracted from file "mutex.adb"

```
 Function
 ========

 This file contains the body for the MUTEX package. This provides a
 mutual exclusion semaphore emulation;

----------------------
package body MUTEX is
----------------------


   task body SEMAPHORE is


      Assume, by default, the resource is not in use.

       Begin infinite loop

          Await a call to seize or release a resource.

             If resource is flagged as not 'in use'

                allow acceptance of a seize resource request

                   and set flag as 'in use'

             If resource is flagged as 'in use'

                allow acceptance of a release resource request

                   and set flag as not 'in use'
```

#### *6.3.2.26 nhk.ads*

Extracted from file "nhk.ads"

```
    Function
    ========

    This file contains the specification for package NHK.

    The function of this package is to provide routine(s) to construct and
    place Non-Periodic Housekeeping (NHK) packets into the telemetry queue
    prior to their being transmitted to the ground.


    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

--------------
package NHK is
--------------


   procedure PUT(SUB_TYPE : PACKET.TELEMETRY_SUBTYPE;
                 SID_EX   : PACKET.SID_TYPE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER);


    The procedure PUT constructs and places an NHK packet in the telemetry
    queue. The interface is as follows:

    where:

    SUB_TYPE specifies the sub-type of NHK packet to be placed in the queue.
             It will take one of the the following values:

             PACKET.EVENT_REPORT          := 1;
             PACKET.EXCEPTION_REPORT      := 2;
             PACKET.MAJOR_ANOMALY_REPORT  := 3;

    SID_EX   specifies the Structure Identifier (SID) to be loaded into the
             packet

    PARAMS   specifies an array of parameters to be loaded into the packet.
             Note - the index range of the parameter array should start at 0.

    SIZE     specifies the number of parameters to be loaded from PARAMS.
```

#### *6.3.2.27 nhk.adb*

```
Extracted from file "nhk.adb"


 Function
 ========

 This package body implements the body for package NHK.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------
package body NHK is
   -------------------


   procedure PUT(SUB_TYPE : PACKET.TELEMETRY_SUBTYPE;
                 SID_EX   : PACKET.SID_TYPE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER) is


        Create an instance of the NHK Packet Data Structure.

       If this packet's SID is enabled

          Place current time in data field header

          Flag presence or absence of CRC in data field header

          Calculate and load packet length

          Load in Structure Identifier (SID)

          Load Number of Parameters

          Load parameters into packet

          Put packet record into queue
```

#### ***6.3.2.28 packet.ads***

Extracted from file "packet.ads"

```
    Function
    ========

    This file contains the specification only package PACKET. This
    defines the format of the telecommand and telemetry packets used by the OM
    instrument and are derived from the description in the 'Telecommand
    and Telemetry Specification', XMM-OM/MSSL/ML/0010.
```

#### *6.3.2.29 peek_poke.ads*

Extracted from file "peek_poke.ads"

```
    Function
    ========

    This file contains the specification for the XMM-OM low-level memory read/write.
    The program is written in assembler and linked as foreign.
```

#### 6.3.2.30 peek_poke.asm

```
File is peek_poke.asm

    Name
     peek

    Description
       Picks up an address to be peeked and the Address State from the stack,
       switches to that Address State, peeks the address, selects the
       original Address State and exits with the value peeked in r2.

    Calling sequence
       var := peek(address,address_state)

       (All parameters & return type are UINT16)

    Input
       r0      Link register
       r2      Uplevel register (not needed ?)
       r14     Frame pointer (not needed ?)
       r15     Stack pointer

    Output
       r2      Holds contents of address peeked

    Altered
       r1, r2, r3, r4

    Register map
       r0      Link register
       r1      Holds entry Address State
       r2      Return value
       r3      Holds address to peek
       r4      Holds Address State to switch to

    Notes
       Assembled for use as a foreign code segment in Ada.
       Registers r0-r4 can be trashed.
       All other registers must be preserved.

    Assumptions

       No error checking is performed.

peekaddr

           Save the current address state and change address state
           Read the memory location
           Restore old address state
           Return


    Name

poke

    Description

Picks up an address to be poked, the Address State and the value

to be poked into memory from the stack, switches to that Address

State, pokes the address, selects the original Address State and

exits with the value poked in r2.

    Calling sequence

var := poke(value,address,address_state);

(All parameters & return type are UINT16)
```

Input

r0

Link register

r2

Uplevel register (not needed ?)

r14

Frame pointer (not needed ?)

r15

Stack pointer

Output

r2

Holds value poked into memory

Altered

r1, r2, r3, r4

Register map

r0

Link register

r1

Holds entry Address State

r2

Holds value to poke and return value

r3

Holds address to poke

r4

Holds Address State to switch to

Notes

Assembled for use as a foreign code segment in Ada.

Registers r0-r4 can be trashed.

All other registers must be preserved.

Is a function because procedure definition in Ada appears

not to link properly (doesn't see assembler label).

Assumptions

No error checking is performed.

pokeaddr

```
Save current address state

Write address with value

Change back to original address state

Return
```

#### 6.3.2.31 rbi.ads

Extracted from file "rbi.ads"

```
    Function
    ========

    This file contains the specification for the RBI package. This, in turn,
    contains RBI service routines. The package RBI and RBI_INT together
    control and monitor the RBI (Remote Bus Interface).

    The code in this package is based on the description of the RBI chip
    given in "Standard RBI Chip For OBDH Interface
    (MC1031 Technical Informations 2.8-01/06/95 and from the
    "OBDH Bus Protocol Requirements Specification", XM-IF-DOR-0002.

--------------
package RBI is
   --------------


   procedure INIT;


       Performs RBI package initialisation.

   function UNCORRECTED_OBT return OBT_TYPE;


       Returns the uncorrected OBT (On-board Time) from the RBI.

   function CORRECT_OBT(UNCORRECTED_OBT_VALUE : in OBT_TYPE) return OBT_TYPE;


       Applies the correction to the OBT documented in the ADV technical note
       2.8-01/06/95

   function CORRECTED_OBT return OBT_TYPE;


       Combines the functions of UNCORRECTED_OBT and CORRECT_OBT;

   procedure SET_OBT(OBT_VALUE : in OBT_TYPE);


       Sets the RBI OBT value. This is usually extracted from an Add Time Code
       packet TM(10,3).

   function "+"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE;


       Adds OBTs together N.B. only accurate to 2**-8 secs!!!!

   function "-"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE;


       Subtract OBTs N.B. only accurate to 2**-8 secs!!!!
       Watchdog Control

   procedure SET_SYNC_READY(SYNC_ENABLE : BOOLEAN);


       Set/Unset Sync Enable Bit in RBI Configuration Register

   task type WATCHDOG_TYPE is
      pragma PRIORITY(IMPORTANCE.CPU_RESET);

      entry PARAMS(TIMOUT : UINT16 ;
                   RESET_INTERVAL : UINT16 ;
                   OK : in out BOOLEAN);
      entry ENABLE;
      entry DISABLE;

   end WATCHDOG_type;


      This task controls the RBI watchdog.

      ENABLE  starts the task.
      DISABLE stops the task.
      PARAMS resets the time intervals used to control the watchdog.
```

```
      TIMOUT specifies what value should be loaded into the
             watchdog timer counter.
      RESET_INTERVAL specifies how often the software the software should
             reload the time counter with TIMOUT.

function TM_READY return BOOLEAN;


    Returns whether TM_READY (telelemetry ready to transmit)  bit is set
    in the RBI status register

procedure SET_TM_READY(SET_TO_ON : BOOLEAN);


    Set/Unset TM_READY (telelemetry ready to transmit) bit in the
    RBI status register

procedure TOGGLE_TM_READY;


    Toggles TM_READY (telelemetry ready to transmit) bit in the
    RBI status register

function TC_READY return BOOLEAN;


    Returns whether TC_READY (ready to receive telecommand) bit is set
    in the RBI status register

procedure SET_TC_READY(SET_TO_ON : BOOLEAN);
pragma INLINE(SET_TC_READY);


    Set/Unset TC_READY (ready to receive telecommand) bit in status register

procedure SET_COMM_AREA_TM_INFO(START_ADDRESS : UINT16;
                                PACKET_LENGTH : UINT16);


    Store start address and length of a telemetry packet in
    the communications area (CCA).

procedure SET_COMM_AREA_TC_INFO(START_ADDRESS : UINT16);


    Store start address of where the telecommmand should be stored
    in the communication area (CCA).

function STATUS_REGISTER return UINT16;


    Returns the RBI Status Register

function CONFIG_REGISTER return UINT16;


    Returns the RBI configuration register
```

#### *6.3.2.32 rbi.adb*

Extracted from file "rbi.adb"

```
 Function
 ========

 This file contains the body for the RBI package. This, in turn,
 contains RBI service routines. The package RBI and RBI_INT together
 control and monitor the RBI (Remote Bus Interface).

 The code in this package is based on the description of the RBI chip
 given in "Standard RBI Chip For OBDH Interface
 (MC1031 Technical Informations 2.8-01/06/95 and from the
 "OBDH Bus Protocol Requirements Specification", XM-IF-DOR-0002.

-------------------
package body RBI is
   -------------------


    Contents of OBT as follows:

         -------------------------------
         | OBT 0    |  OBT 1  |  OBT 2  |  OBT location
         -------------------------------
         |    C     |    D    |    E    |  Register
         -------------------------------
         |0       15|16     31|32-42|xxx|  Bits in Counter
         -------------------------------
         |    SECS       |     FRAC     |  Secs/Fractions of sec
         -------------------------------
         |23            0|-1     -19|xxx|  2**? secs
         -------------------------------

     Note the layout of the SCET in a packet for comparison (and its offset)

          23            0 -1     -16
    -------------------------------
    |    Coarse Time     |  Fine   |
    -------------------------------

   function UNCORRECTED_OBT return OBT_TYPE is


       Ensure exclusive use of RBI configuration register
       while we peform a Freeze operation.

      "Freeze" the current time by writing appropriate instruction
       to the RBI configuration register.

       Release the register for use by other code.

       Read and store bits 0-15 of the result.

       Read bits 16-31 of the result

       Read remaining bits 32-42 (result in high order bits)

      Return the stored result (i.e. the OBT as defined above).

   function CORRECT_OBT(UNCORRECTED_OBT_VALUE : in OBT_TYPE) return OBT_TYPE is


       if bits 32 to 42 of the counter freeze 2 is greater than 3ff hex

          subtract 1 from bits 0 to 31

          Otherwise

            subtract one from 2nd word

      Return the result (a corrected OBT).

   function CORRECTED_OBT return OBT_TYPE is


       Get the OBT and correct it.

   procedure SET_OBT(OBT_VALUE : in OBT_TYPE) is
```

```
    Prevent use of Freeze register while we do this.

    Write the most significant 16 bits of the provided OBT
    into the 1st RBI OBT update register

    Write the next 16 bits of the provided OBT
    into the 2nd RBI OBT update register

    Release Freeze register

function "+"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE is


   Prevent Overflows on addtions.

    Convert the OBT's to long integers, add and convert back.

function "-"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE is


   Prevent Overflows on subtractions.

    Convert the OBT's to long integers, subtract and convert back.

function TO_OBT_TYPE(INPUT : in LONG_INTEGER) return OBT_TYPE is


   This routine is used internally to the package to convert
   a supplied 64 bit integer into an OBT format (3*16 bit words).

function TO_LONG_INT(INPUT : in OBT_TYPE) return LONG_INTEGER is


  This routine is used internally to the package to convert
   a supplied OBT (3*16 bit words) into a 64 bit integer.

procedure SET_SYNC_READY(SYNC_ENABLE : BOOLEAN) is


    Get the RBI configuration register value

    If the Synchronisation Enable bit is not as required

       Toggle it

task body WATCHDOG_TYPE is


    Begin infinite loop

       Await a call to one of the rendevous points

          If a call to the set params entry point is made

            Remember the specified timout period (units = 1/256 secs )
            and reset interval

            Flag as valid.

          If a call to enable the watchdog is made

             Determine if watchdog is already enabled

             Write timout period to appropriate register

             If necessary, enable watchog

          If a call to disable the watchdog is made

             Determine if watchdog is enabled

             If so, disable it

       OR

          Provided the watchdog is enabled

             and if no call to a rendevous is made for reset period

             Reset counter in watchdog (thus as long as the ICU code
             is running, the timout counter is never allowed to get
             to zero.
```

```
procedure INIT is

     Set up the comms area by writing appropriate values to registers

    Ensure TC and TM ready flags are disabled for now

function TM_READY return BOOLEAN is

    Get the RBI Status register value

    Extract and return the TM_READY bit

procedure SET_TM_READY(SET_TO_ON : BOOLEAN) is

    If the telemetry ready for transmission (TM_READY) bit is  not
    already in the requested status

       Toggle it so it is

procedure TOGGLE_TM_READY is

   Toggle the current RBI TM_READY (telemetry ready for transmission)
   bit state

function TC_READY return BOOLEAN is

    Get RBI status register value

    Extract and return the TC_READY
    (ready to receive a telecommand) bit

procedure SET_TC_READY(SET_TO_ON : BOOLEAN) is

    Get current status RBI register.

    If bit 11 (the TC_READY- ready to receive a telecommand) is
    already in the required status

       Do nothing

       Otherwise if it needs to be on

         Set it on in the RBI status read back earlier

       else

         Clear it in RBI status read back earlier.

       Finally, write back the resulting RBI status word to the
       register (NOTE: only bits 11-15 can be written to)

procedure SET_COMM_AREA_TM_INFO(START_ADDRESS : UINT16;
                                PACKET_LENGTH : UINT16) is

    Store the start address of the TM packet in bytes,
    relative to the start adddress of the CCA, in the CCA,

    Store the packet length in the CCA in words but
    with 1 subtracted and the MSB set, as per specification.

procedure SET_COMM_AREA_TC_INFO(START_ADDRESS : UINT16) is

    Store in TC packet start address in bytes relative to the start
    of the CCA, in the CCA.

function CONFIG_REGISTER return UINT16 is

    Get the config register value

function STATUS_REGISTER return UINT16 is

    Get the status register value
```

#### *6.3.2.33 rbi_ih.ads*

Extracted from file "rbi_ih.ads"

```
    Function
    ========

    This file contains the specification for the XMM-OM rbi interrupt handler.
    The interrupt handler is written in assembler and linked as foreign.
```

#### ***6.3.2.34 rbi_ih.asm***

File is rbi_ih.asm

```
        This follows closely the document:
        OBDH Bus Protocol Requirement Specification
        XM-IF-DOR-0002
          Fetch the interrupt counter
          Check for impending overflow
          If it's OK, increment it
          otherwise avoid overflow
          read config_reg
          get the bits we're interested in
          is it lossn (0)?
          is it instruction to user (1)?
          is it instruction to rbi (2)?
          is it other_it (3)?


          otherwise serious error so safe


          Read value from appropriate register
          (which also clears the interrupt)
          read instruction to user reg
          If the register is 0, jump to tcq_add
          when it's an Instruction to RBI interrupt


          read instruction to rbi reg
          This could be caused by warm reset and we
          call back into the bootstrap (TBI)


          If it's any other sort of interrupt
          This is an error (so we safe or discard with exception, TBD)
          and finish off
         -------------------------------------------------------------------
          tcq_add **********************************************************
         -------------------------------------------------------------------
          set tc_ready to false
          if full
            Tell s/c we can't accept packets (This ought never happen as we take packets away in
time?)
            read input_pointer from memory
            add one
            mod it with no_tc_slots
            keep for future
            store it again
            Now set up new address for next packet
            start_address = 16#404# + r0*248
          if not tc_q.is_full
          i.e.
          if (input_pointer+1)&3 != output_pointer
            (increment input_pointer)
            the required mask is 0
          else required mask = set_tc_ready_mask (16#0010#)
            Read status
            'and' this status with set_tc_ready_mask (16#0010#);
            Compare this with the required mask
            If they're the same, finish off
            if REQUIRED_MASK = SET_TC_READY_MASK (16#0010#)
              'or' the status that was read with set_tc_ready_mask (16#0010#)
            else 'and' the status that was read with clear_tc_ready_mask (16#ffef#)
              xio this to the rbi_status reg
              finish off
          Read status
          If the tm_ready bit is set
            write a reset output transfer request to the rbi config reg
          Increment the output_pointer
          Read the input_pointer and compare output_pointer with input_pointer
          If they're equal
            finish off
          Otherwise calculate the address and write it to cca_tm_start
          Calculate the length and write it to cca_tm_length
          Read the RBI status
          'and' it with the tm_ready_mask (16#0080#)
          finish off
          if zero, write a reset_output_transfer_request to the RBI config reg
          finish off
          Tidy up after finishing
          FINISH OFF:
```

```
Recover registers
Turn on interrupts
Back from whence we came
```

#### *6.3.2.35 reset.ads*

Extracted from file "reset.ads"

```
    Function
    ========

    This file contains the specifications for the XMM-OM reset package.
    reset is written in assembler and linked as a foreign.

----------------
package RESET is
----------------


procedure RESET(PARAM : UINT16);


    This procedure changes the mode of the ICU.
```

#### *6.3.2.36 reset.asm*

```
File is reset.asm

    Name
     reset

    Description
       When called, enables the start up ROM and jumps to
       location zero.
       Disable interrupts
       Stop timer B
       Make sure we are in address state 0
       Copy new interrupt vectors to data space
       Copy new interrupt vectors to instruction space
       Reselct page 0
       Clear all interrupts
       Now start op code
       Now start operational code
```

#### 6.3.2.37 ssi_driver.ads

Extracted from file "ssi_driver.ads"

```
   ------------------------
   procedure SSI_INTERRUPT;
   ------------------------


       SSI_INTERRUPT is the SSI interrupt handler (written in Ada but
       connected via the assembly code ssi_ih.asm)

   ----------------
   procedure RESET;
   ----------------


       This procedure resets the SSI link
      (software only---there is no hardware reset)

   SSI_ERROR_COUNT : UINT16 := 0;


       This variable is a counter for the number of SSI errors that have occured

   HEARTBEAT_COUNTER : UINT16 := 0;


       This variable is a counter for the number of heartbeats that have occured
       It wraps at 0xffff back to 0 then 1 etc.

   SSI_INT_COUNT : UINT16 := 0;


       This variable is a counter for the number of SSI interrupts received
       It wraps back to 0 after 0xffff
```

#### *6.3.2.38 ssi_driver.adb*

**Extracted from file "ssi_driver.adb"**

```
 Function
 ========

 This file contains the body for package ssi_driver.
 It writes to and reads from the SSI interface.

 Reference
 =========

 The SSI interface is described in a document.

 Dependencies
 ============

with SYSTEM;
with UNCHECKED_CONVERSION;
with INTRINSICS ; use INTRINSICS;
with ARTCLIENT;

with DEBUG;
with MEMLOC;


 Suppress all checks to speed up

--------------------------
package body SSI_DRIVER is
   --------------------------


    The first word of an SSI block read back by the ssi_ih interrupt handler
    is stored at MEMLOC.SSI_FIRST_WORD_LOCATION for speed.

   procedure SSI_INTERRUPT is


       This (Ada code) is called from ssi_ih.asm (assembler code)

       interrupts are already disabled by the 31750's microcode

       - Read Data -

       Read first word of SSI block from the special address that
       the assembler code (ssi_ih) wrote to

       remember the initial timer B value

       Turn on RBI interrupts

       loop

          get the SSI status

          If there's more data to read - read it

             if the count of words in this block gets far too large, store an error

                otherwise increment the READ count

             reset the old stored value of timer B because we haven't stopped receiving data yet

             but if there's nothing to read this time round
             check the timer

             if timer B has wrapped round, add on 64K

             exit the loop when we've been waiting to read something for 40 timer-B ticks (4 ms)

          read the SSI status

          if there's been an overflow

             clear the overflow

             do a dummy read to clear

             store an error "-8"
```

```
   end loop

get the second word of the SSI block from the output buffer
this contains the number of words minus two that should be in the block
if the number read is just too large

   remember an error "-11"

read the SSI status

if there's been an overflow

   clear the overflow

   do a dummy read to clear

   store an error "-7"

clear SSI interrupt by writing to the SSI interface
```

#### *6.3.2.39 ssi_ih.ads*

Extracted from file "ssi_ih.ads"

```
    Function
    ========

    This file contains the specification for the XMM-OM ssi interrupt handler.
    The interrupt handler is written in assembler and linked as foreign.
```

#### *6.3.2.40 ssi_ih.asm*

File is ssi_ih.asm

```
         Sort out the stack
         Read first word of SSI block from DPU to ICU and store for Ada
         Jump to Ada SSI interrupt handler
           Tidy up
    Beware of strange arithmetic (eliminate complaints)
    Prohibit preemption
    Recover R15 contents
    Release interrupt stack
       Recover register R15
       Recover registers R0 to R3
           Return from interrupt
```

#### 6.3.2.41 task_report.ads

Extracted from file "task_report.ads"

```
    Function
    ========

    This file contains the specification for package TASK_REPORT.

    The function of this package is to provide routine(s) to construct and
    place Task Parameter Report packets into the telemetry queue
    prior to their being transmitted to the ground.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

----------------------
package TASK_REPORT is
----------------------


   procedure PUT(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER);


       The procedure PUT constructs and places a Task Param Report packet
       associated with TID and FID
       in the telemetry queue. The interface is as follows:

       where:


       PARAMS   specifies an array of parameters to be loaded into the packet.
                Note - the index range of the parameter array should start at 0.

       SIZE     specifies the number of parameters to be loaded from PARAMS.
```

#### *6.3.2.42 task_report.adb*

Extracted from file "task_report.adb"

```
 Function
 ========

 This file contains the body for package TASK_REPORT.

 The function of this package is to provide routine(s) to construct and
 place Task Parameter Report packets into the telemetry queue
 prior to their being transmitted to the ground.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------
package body TASK_REPORT is
-------------------


   procedure PUT(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER) is


       Flag presence or absence of CRC in data field header

       Calculate and load packet length

       Load parameters into packet

       Put packet record into queue
```

#### *6.3.2.43 taskman.ads*

Extracted from file "taskman.ads"

```
    Function
    ========

    This package contains the specification for the TASKMAN package.
    The function of this package is to interpret the Task
    Management Telecommands and forward them to the appropriate code.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

------------------
package TASKMAN is
------------------


   function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN;


       The function REQUEST provides the means of passing the telecommand
       to the package for action.

       where:

       TC_PACKET contains the packet to be interpreted and executed.
```

#### *6.3.2.44 taskman.adb*

Extracted from file "taskman.adb"

```
 Function
 ========

 This package contains the body for the TASKMAN package.
 The function of this package is to interpret the Task
 Management Telecommands and forward them to the appropriate code.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-----------------------
package body TASKMAN is
   -----------------------


   function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


       Set up default error condition of command not being accepted.

       Select action on the basis of packet subtype.

          When the packet subtype is Start Task...

             Set up default error of illegal TID

             Select Action on the basis of the Task Identifier (TID)

             if its a normal TMPSU normal heater configuration command

                Turn on 1 heater

                Wait a bit

                then turn on 2nd heater

                Flag as accepted

             If its a secondary voltages command

                Enable them and flag as accepted

             If its a DEMPSU reset

               Reset/Turn-on the DPU

                And flag as accepted

             If its a watchdog command

                Enable it

                and flag as accepted

             If its an HK command

                Start it

                and flag as accepted

             If it's an ICB Direct command.

                Allow direct writing to the ICB

                and flag as accepted.

             when TID is any other value

             End of selection

          When the packet subtype is Stop Task...

             Set up default error of illegal TID
```

```
   Select Action on the basis of the Task Identifier (TID)

   If it's a TMPSU heater command

      Turn off one heater

      Wait a bit

      then turn off the other heater

      Flag as accepted

   If it's a secondary voltage command

      Disable them and flag as accepted

   If it's a watchdog command

      Disable it

      and flag as accepted

   If it's an HK command

      Disable it

      and flag as accepted

   If it's an ICB Direct Command

      Disallow direct writing to the ICB ports

      and flag command as accepted.

  when TID is any other value --------------------------------

      Flag as invalid task

      End of Selection

When the packet subtype is Load Task...

   Set up default error of illegal FID

   Select Action on the basis of the Task Identifier (TID)

   when it's a  ICB Direct command

         and the FID value indicates a write to an ICB port.

            and direct writing to ICB ports is enabled

               Output supplied datum to specified
               address and subaddress

               In this code, always flag as accepted.

            Otherwise

               Issue an Unsuccessful Acceptance Packet

               and flag command as unaccepted.

         Any othe value of FID

            Flag it as an invalid command.

   If it's a watchdog command

         If the FID indicates a watchdog timout class of command

            Reset the controlling parameters

         Otherwise

            Flag as an invalid command

   when TID is any other value

      Flag as a coomand error of illegal an TID

      End of Selection
```

```
    When the packet subtype is Report Task...

       Set up default error of illegal FID

       Look at the TID

          If it's a valid read ICB port type

             and direct access to the ICB is enabled

                Request the appropriate task report packet
                and flag as an accepted command

             otherwise

                Issue an unsuccessful acceptance packet.

                and flag as such

          Otherwise

             Flag as an illegal comand with a TID error

       When the packet subtype is Mode Transition...

       Set up a default error of illegal mode

       Then perform change to operational mode via the Mode Manager
       code.

 If the supplied command was an invalid task management command,

    inform the ground with an Unsuccessful Acceptance Command packet.

 Return success only if we had both a valid task command and
 it was not rejected by called functions as a bad command.
```

***6.3.2.45 tc_q.ads***

Extracted from file "tc_q.ads"

```
    Function
    ========

    This file contains the specification for the package TC_Q. That package
    supplies the routines that manipulate the telecommand queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010
    The OBDH protocol is defined in XM-IF-DOR-0002

---------------
package TC_Q is
---------------


       Define number of slots NO_SLOTS in Telecommand Queue

       Define telecommand queue data structure as follows

             Description                  Size (Words)
             ===========                  ============

       ********************************
       *      Packet Slot 0           *       124
       *------------------------------*
       *    ... and so on until...    *       124
       *------------------------------*
       *      Packet Slot n-1         *       124
       ********************************

       Two pointers are used to indicicate the 'occupation' of the queue.

       The Input Pointer indicates the packet slot into which the
       the next packet will be written.

       The Output Pointer indicates the packet slot from which the
       the next packet should be taken.

       In addition, there is a communication area which the spacecraft examines
       to determine the location of a TM packet to be collected or into which
       a TC packet should be loaded.

       *************************************
       *     RBI Status Word               *
       *-----------------------------------*
       * Start Address of TM Source Packet *
       *-----------------------------------*
       *     Length of TM Source Packet    *
       *-----------------------------------*
       * Start Address of TC Source Packet *
       *************************************

       Create instance of Q data structure, and fix at location in memory

        Define the input and output pointers at a fixed location in memory.

   procedure RESET;


       This procedure resets (i.e. clears) the TC queue

   procedure REMOVE(PCKT : in out PACKET.TC_TYPE );


       This procedure removes a packet from the TC queue

       where:

       PCKT is the packet removed from the TC queue.

   procedure ADD;


       This procedure informs the ICU that the s/c had DMAd a TC packet
```

```
    NOTE: This routine is now obsolete and should have been removed.
    Its function is now handled by a low level assembler routine
    in package RBI_IH.

function IS_EMPTY return BOOLEAN;


    This function determines whether the TC queue is empty
    It returns TRUE if the queue is empty

function IS_FULL return BOOLEAN;


    This function determines whether the TC queue is full
    It returns TRUE if the queue is full
```

#### *6.3.2.46 tc_q.adb*

```
Extracted from file "tc_q.adb"


 Function
 ========

 This file contains the body for the package TC_Q. It
 supplies the routines that manipulate the telecommand queue directly.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010.
 The OBDH protocol is defined in XM-IF-DOR-0002

--------------------
package body TC_Q is
--------------------


   Define telecommand queue data structure as follows

         Description                  Size (Words)
         ===========                  ============

   ********************************
   *      Packet Slot 0           *       124
   *------------------------------*
   *    ... and so on until...    *       124
   *------------------------------*
   *      Packet Slot n-1         *       124
   ********************************

   Two pointers are used to indicicate the 'occupation' of the queue.

   The Input Pointer indicates the packet slot into which the
   the next packet will be written.

   The Output Pointer indicates the packet slot from which the
   the next packet should be taken.

   In addition, there is a communication area which the spacecraft examines
   to determine the location of a TM packet to be collected or into which
   a TC packet should be loaded.

   *************************************
   *     RBI Status Word               *
   *-----------------------------------*
   * Start Address of TM Source Packet *
   *-----------------------------------*
   *     Length of TM Source Packet    *
   *-----------------------------------*
   * Start Address of TC Source Packet *
   *************************************

   procedure RESET is


       Set the start and end pointers to the 1st packet

       Store the Start address of the 1st packet in the comm area

       Inform s/c we are ready to receive a packet by setting the
       appropriate RBI status word bit.

   procedure REMOVE(PCKT : in out PACKET.TC_TYPE ) is


       Copy packet from current slot

       calc next pointer value

       Inform s/c we are ready to receive a packet again by setting the
       appropriate RBI status word bit (provided the queue is not full).

   procedure ADD is


       NOTE: This routine is now obsolete and should be removed.
```

```
    Its function is now handled by a low level assembler routine in
    package RBI_IH.

    Tell s/c we can't receive TC packets

    Packet has already been stored by s/c
    So calculate next slot index

    Now set up new address for next packet

    Now tell s/c we can accept TC packets again if q not full

function IS_EMPTY return BOOLEAN is


    Return TRUE if Input Pointer equals the Output Pointer

    otherwise return FALSE

function IS_FULL return BOOLEAN is


    calc index of next (after current) packet slot to be written

    return TRUE if same as next location to be read
```

#### *6.3.2.47 tc_verify.ads*

Extracted from file "tc_verify.ads"

```
    Function
    ========

    This file contains the specification for the TC_VERIFY package.

    That package supplies the routines that construct and send the
    telecommand verification packets.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

--------------------
package TC_VERIFY is
--------------------


   procedure SUCCESSFUL_ACCEPTANCE
     (TC_SEQ_COUNT_AND_SRC: UINT16);


       This procedure constructs and sends a successful telecommand acceptance
         packet to the telemetry queue.

       where:

       TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                            telecommand being verified.

   procedure UNSUCCESSFUL_ACCEPTANCE
     (TC_SEQ_COUNT_AND_SRC: UINT16;
      ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
      NO_PARAMS           : UINT16;
      PARAMS              : UINT16_ARRAY);


       This procedure constructs and sends an unsuccessful telecommand
       acceptance packet to the telemetry queue.

       where:

       TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                            telecommand being verified.

       ERROR_CODE           specifies the reason for failure

       PARAMS               specify any parameters associated with the
                            error code (NOTE - unlike other routine in the
                            ICU code, the first index of this array must be 1)

   procedure UNSUCCESSFUL_EXECUTION
     (TC_SEQ_COUNT_AND_SRC: UINT16;
      ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
      NO_PARAMS           : UINT16;
      PARAMS              : UINT16_ARRAY);


       This procedure constructs and sends an unsuccessful telecommand
       execution packet to the telemetry queue.

       where:

       TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                            telecommand being verified.

       ERROR_CODE           specifies the reason for failure

       PARAMS               specify any parameters associated with the
                            error code (NOTE - unlike other routine in the
                            ICU code, the first index of this array must be 1)
```

#### *6.3.2.48 tc_verify.adb*

Extracted from file "tc_verify.adb"

```
 Function
 ========

 This file contains the body for the TC_VERIFY package.

 That package supplies the routines that construct and send the
 telecommand verification packets.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------------
package body TC_VERIFY is
-------------------------


    The specification for this package's internal routine follows:
    ==============================================================

   procedure UNSUCCESSFUL(
                          SUB_TYPE            : PACKET.TELEMETRY_SUBTYPE;
                          TC_SEQ_COUNT_AND_SRC: UINT16;
                          ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                          NO_PARAMS           : UINT16;
                          PARAMS              : UINT16_ARRAY);


    where:

    SUB_TYPE             is the packet sub-type being created

    TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                         telecommand being verified.

    ERROR_CODE           specifies the reason for failure

    NO_PARAMS            specifies how many params are supplied

    PARAMS               specify any parameters associated with the
                         error code

    The body for this package's internal routine follows:
    =====================================================

   procedure UNSUCCESSFUL(
                          SUB_TYPE            : PACKET.TELEMETRY_SUBTYPE;
                          TC_SEQ_COUNT_AND_SRC: UINT16;
                          ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                          NO_PARAMS           : UINT16;
                          PARAMS              : UINT16_ARRAY) is


       Create verification packet of requested sub-type

       Get the time and place it in packet

       Flag CRC as present

       Store the number of parameters supplied

       Calculate and load packet length

       Copy originating sequence count and source flag into packet

       Copy error code into packet

       and then copy in the associated parameters

       Place packet in queue

    The bodies for this package's externally visible routines follow:
    =================================================================

   procedure UNSUCCESSFUL_EXECUTION
```

```
  (TC_SEQ_COUNT_AND_SRC: UINT16;
   ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
   NO_PARAMS           : UINT16;
   PARAMS              : UINT16_ARRAY) is


    Call UNSUCCESSFUL with sub-type specifying Unsuccessful Execution

procedure UNSUCCESSFUL_ACCEPTANCE
  (TC_SEQ_COUNT_AND_SRC: UINT16;
   ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
   NO_PARAMS           : UINT16;
   PARAMS              : UINT16_ARRAY) is


    Call UNSUCCESSFUL with sub-type specifying Unsuccessful Acceptance

procedure SUCCESSFUL_ACCEPTANCE
  (TC_SEQ_COUNT_AND_SRC: UINT16) is


    Create verification packet of sub-type Succesful Acceptance

    Get the time and place it in packet

    Flag CRC as present

    Calculate and load packet length

    Copy originating sequence count and source flag into packet

    Place packet in queue
```

#### *6.3.2.49 tcq.ads*

Extracted from file "tcq.ads"

```
    Function
    ========

    This file contains the specification for the package TCQ.
    That package supplies the low level routines that manipulate the
    telecommand queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010.
    The OBDH protocol is defined in XM-IF-DOR-0002.

--------------
package TCQ is
--------------


   procedure RESET;


       This procedure resets (i.e. clears) the telecommand queue


   procedure GET(PCK         : in out PACKET.TC_TYPE;
                 GOOD_PACKET : out BOOLEAN);


       This procedure returns the next valid telecommand packet received
         to the caller

       where:

       PCK   is the returned packet.

       GOOD_PACKET - always returns TRUE.

   procedure ADD renames TC_Q.ADD;


       The procedure is called when an EOTC Instruction to User
       interrupt is received (i.e. that a TC packet has been added to the
       TC queue).
       NOTE: This routine is now obsolete and should have been removed.
       Its function is now handled by a low level assembler routine
       in package RBI_IH.
```

#### *6.3.2.50 tcq.adb*

Extracted from file "tcq.adb"

```
 This package body implements the specification given in TCQ.ADS

 Dependencies
 ============

with TC_Q;
with TMQ;
with TC_VERIFY;
with TYPES; use TYPES;
with CRC;
with HK;
with DEBUG;


-------------------
package body TCQ is
   -------------------


    Data Global to this package
    ===========================

    As this package only returns valid packets, it holds a table
    of types and subtype, and any associated error conditions,
    as follows:

        Subtype 0 1 2 3 4 5 *  Comments
    Type
      1         ? ? ? ? ? ? ?
      2         I o o I I I I  Device Commanding
      3         ? ? ? ? ? ? ?
      4         ? ? ? ? ? ? ?
      5         I o o o o o I  Task Management
      6         I o o o I I I  Memory Maintenance
      7         ? ? ? ? ? ? ?
      8         ? ? ? ? ? ? ?
      9         I o I o o o I  Telemetry Maintenance
     10         I I o o I o I  Time Management
     11         ? ? ? ? ? ? ?
     12         ? ? ? ? ? ? ?
     13         I o I I I I I  Test Commands
     14         ? ? ? ? ? ? ?
     15         ? ? ? ? ? ? ?

    where:

    o = valid type/subtype, i = invalid subtype, ? = invalid type

   function VALID_PACKET(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


       If a good packet

          Perform Valid APID check

             If not, note and flag it

       If still a good packet

          Perform Packet Length Check (is it in a valid range)

             If not, note and flag it

       If still a good packet

          Perform CRC check

          If the CRC check fails

             Note and flag it

       If still thought to be OK

          Look up error condition, if any, as a function of packet type
          and subtype, from the table described above.

          Select next action on the basis of the value returned.
```

```
        When packet OK

           Return a value of TRUE

           When an invalid packet is present

           Determine correct error code

           Load up the packet type and subtype into the parameter
           array

           Finally flag as bad

    If it's not a good packet so far

       Construct and place Unsuccessful Acceptance
       Telemetry Packet in the telemetry queue.

       and count the bad packets

    Return status of packet

procedure RESET is


    Perform Reset of the TC queue.

procedure GET(PCK         : in out PACKET.TC_TYPE;
              GOOD_PACKET : out BOOLEAN) is


    Commence loop

       If the telecommand queue is empty

          then wait a while

       otherwise

          Remove a packet from the queue

          Use function VALID_PACKET to check the packet.
          If it returns a value of TRUE
          (i.e. we have a valid packet).

             then exit from this procedure, indicating success

       End Loop
```

#### *6.3.2.51 time_man.ads*

Extracted from file "time_man.ads"

```
    Function
    ========

    The file contains the specification for the Time Manager Package TIME_MAN.
    This package, together with the package BCP4_IH, supplies routines to
    support On-Board Time Management.

-------------------
package TIME_MAN is
   -------------------


   function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN;


      This routine implements the On-Board Time Management Packets TC(10,x)
      contained in TC_PACKET. The format of these packets is defined in
      the Packet Structure Definition document PX-RS-0032. Of those, only
      the following are required to be supported.

      TC(10,2) - Enable Time Synchronization.
      TC(10,3) - Add Time Code.
      TC(10,5) - Enable Time Verification.

      In this release, the function always returns TRUE.

   function VERIFICATION_ACTIVE return BOOLEAN;


       Returns TRUE if time verification is active

   function SYNCHRONISATION_ACTIVE return BOOLEAN;


       This function returns TRUE if the process of synchronizing the time
       is in progress.

   function TIME_STAMP return PACKET.TIME_TYPE;


       This function returns the current on-board time in a format suitable
       for direct insertion into a packet.
       (see the RBI package for details of the format).
```

#### *6.3.2.52 time_man.adb*

Extracted from file "time_man.adb"

```
 Function
 ========

 The file contains the body for the Time Manager Package TIME_MAN.
 This package, together with the package BCP4_IH, supplies routines to
 support On-Board Time Management.

------------------------
package body TIME_MAN is
------------------------


   function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


       Determine action on the basis of the packet sub-type.

         If we have received a Time Sync Packet

           Inform world that we are synchronising by setting
           the appropriate flag.

           Enable time synchronisation by commanding the
           RBI configuration register appropriately.

         If we have received an Add Time Code Packet

           Remember the most significant byte from the time information
           supplied by the packet.

           Copy remaining significant 4 bytes into work array

           Convert them to RBI OBT (On-Board Time) format and
           load into RBI registers

           Now disable Time synchronisation by commanding the RBI
           configuration register accordingly.

           Finally, tell world we are no longer synchronising by resetting
           the appropriate flag.

           and ensure other flag is set off to indicate time is now valid

         If we have received an Enable Time Verification Packet

           Inform world we are verifying the time by setting the
           appropriate flag

           Start BCP4 processing task (see below)

           and leave it to do the work

         For any other packet sub-types.

           Do nothing.

       In this release, always return success.

   task body BCP4 is


       Begin looping

          Wait until a call to start the task occurs

          Enable BCP4 processing at interrupt level

          then wait for bcp4 int to be processed by code in
          package RBI_IH (i.e. load up the OBT)

          Correct the On Board Time obtained from RBI

             Create instance of a Time Management Report packet

             Now build Time Verification Packet

             Flag CRC as present
```

```
        Calculate and load packet length

        Construct Most Sig Byte of time stamp from value
        extracted from Add Time Code packet and held in memory.

        Construct remaining bytes from corrected OBT

        And send it to to TM queue.

     and disable BCP4 processing

     and inform world we have finished verifying the time.

function SYNCHRONISATION_ACTIVE return BOOLEAN is


    Return the value of the synchronising flag

function VERIFICATION_ACTIVE return BOOLEAN is


    Return the value of the verification flag

function TIME_STAMP return PACKET.TIME_TYPE is


    Construct Most Sig Byte of time stamp from value extracted
    earlier from the Add Time Code packet and held in memory

    Get current corrected On-Board Time from the RBI

    Construct remaining bytes of time stamp from it;

    Return the time stamp.
```

#### *6.3.2.53 tm_man.ads*

Extracted from file "tm_man.ads"

```
    Function
    ========
    This file contains the specification for the telementry manager package, TM_MAN.


    Reference
    =========

    XMM-OM/MSSL/ML/0010.1


-----------------
package TM_MAN is
-----------------


function REQUEST(TM_MAN_PACKET : PACKET.TC_TYPE) return BOOLEAN;


       This function provides the means of passing the telecommand
       to the package for action.

       where :

       TM_MAN_PACKET contains the tc packet to be interpreted and executed.


function SID_STATUS(SID : PACKET.SID_TYPE) return BOOLEAN;


       This function reports on the TM packet generation status of a
       packet with the corresponding packet type specified by SID.

       where :

       SID is the tm packet sid to be reported

       If the generation of a TM packet with this SID is enabled then
       the function will return TRUE, FALSE otherwise.


function REPORT_STATUS(SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN;


       This procedure is responsible for generation of a TM(9,1) packet in
       response to a TC(9,1) packet.

       where :

       SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
       of the associated telecommand.

       Returns TRUE if command was successfully accepted


----  function CHANGE_ALL(ENABLE_DISABLE : BOOLEAN;
----                      SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN;


       This procedure changes the generation status of all applicable
       TM packets to that specified by ENABLE_DISABLE. The
       SEQUENCE_COUNT_AND_SRC parameter is needed in case of unsuccessful
       command execution

----  function CHANGE_SPECIFIC(ENABLE_DISABLE : BOOLEAN;
----                           SID : PACKET.SID_RECORD_ARRAY;
----                           SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN;


       This procedure changes the generation status of the TM packets
       specified by the SID parameter to that specified by ENABLE_DISABLE.
       SEQUENCE_COUNT_AND_SRC parameter is needed in case of unsuccessful
       command execution
```

#### 6.3.2.54 tm_man.adb

```
Extracted from file "tm_man.adb"


 Function
 ========

 This file implements the body of the package TM_MAN for BASIC


 Reference
 =========

 XMM-OM/MSSL/ML/0010.1


----------------------
package body TM_MAN is
----------------------



    Create the enabled array which contains true if a particular
    sid is to be enabled (ie a tm packet with that sid can be
    generated)



    Create the valid array which contains true if a particular
    sid is defined


  function REQUEST(TM_MAN_PACKET : PACKET.TC_TYPE) return BOOLEAN is


         Check whether CRC is present

         Now determine packet subtype and act accordingly

            1 for a Report TM Packet Generation Status

            2 for an enable Generation of all TM Packets

            3 for a Disable Generation of all TM Packets

            4 for an Enable Generation of Specific Packets

            5 for a Disable Generation of Specific Packets

            Any other value return false

  function SID_STATUS(SID : PACKET.SID_TYPE) return BOOLEAN is


         Return the SID value in the valid sid array
         or'ed with the value in the enables array

  function REPORT_STATUS(SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN is


         Loop over the valid sid array, getting all SID enabled status
         and put them in an array making up the data portion of the
         telemetry packet

         Now create rest of the telemetry packet

         Now put complete packet into the tm queue

  function CHANGE_ALL(ENABLE_DISABLE         : BOOLEAN;
                      SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN is


         Loop over the enabled sid array

            Record enabled status in the array

  function CHANGE_SPECIFIC(ENABLE_DISABLE         : BOOLEAN;
                           SID                    : PACKET.SID_RECORD_ARRAY;
                           SEQUENCE_COUNT_AND_SRC : UINT16;
                           PKT_LENGTH             : UINT16) return BOOLEAN is
```

```
Calculate the number of sids to change
If valid number of sids then
   Set up error parameters just in case
   Test whether sid to change is a valid one
   If this is a valid sid
      If enabling this sid
            Determine sid type is
            When fast hk
               If slow hk is already enabled then
                  cannot enable fast hk
            When slow hk
               If fast hk already enabled then
                  cannot enable slow hk
incorrect number of sids
   Cannot change any sids
If status of the sid can be changed then
   Record changed sid status
else
   send unsuccessful acceptance packet
```

#### 6.3.2.55 tm_q.ads

Extracted from file "tm_q.ads"

```
    Function
    ========

    This file contains the specification for package TM_Q.

    That package supplies the low level routines that manipulate the
    telemetry queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010
    The OBDH protocol is defined in XM-IF-DOR-0002

---------------
package TM_Q is
---------------

       Two pointers are used to indicate the 'occupation' of the queue.

       The Input Pointer indicates the packet slot into which the
       the next packet will be written.

       The Output Pointer indicates the packet slot from which the
       the next packet should be taken.

       Define the input and output pointers at a fixed location in memory.

   procedure RESET;

       This procedure resets (i.e. clears) the TM queue

   procedure ADD(PCKT : in PACKET.TM_TYPE );

       This procedure adds a packet to the TM queue

       where:

       PCKT is the packet to be added to the TM queue.

   function IS_FULL return BOOLEAN;

       This function determines whether the TM queue is full

       where IS_FULL returns TRUE if the queue is full

   procedure REMOVE;

       This procedure remove a packet from the telemetry queue after
       the s/c indicates it has taken a copy with an
       EOTM Instruction to User.

       NOTE: This routine should have been removed as its function is now
       performed by a low-level assembler routine in package RBI_IH.

   function PACKET_COUNT return UINT16;

       Returns current packet sequence count.
```

#### 6.3.2.56 tm_q.adb

```
Extracted from file "tm_q.adb"

 Function
 ========

 This file contains the body for package TM_Q.

 That package supplies the low level routines that manipulate the
 telemetry queue directly.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010.

 The OBDH protocol is defined in XM-IF-DOR-0002

--------------------
package body TM_Q is
--------------------


    The telemetry queue is a area of memory defined as follows:

          Description                  Size (Words)
          ===========                  ============

    ********************************
    *      Packet Slot 0           *       259
    *------------------------------*
    *    ... and so on until...    *       259
    *------------------------------*
    *      Packet Slot n-1         *       259
    ********************************

    Two pointers are used to indicate the 'occupation' of the queue.

    The Input Pointer indicates the packet slot into which the
    the next packet will be written.

    The Output Pointer indicates the packet slot from which the
    the next packet should be taken.

    In addition, there is a communication area which the spacecraft examines
    to determine the location of a TM packet to be collected or into which
    a TC packet should be loaded.

    *************************************
    *     RBI Status Word               *
    *-----------------------------------*
    * Start Address of TM Source Packet *
    *-----------------------------------*
    *     Length of TM Source Packet    *
    *-----------------------------------*
    * Start Address of TC Source Packet *
    *************************************

    Create instance of Q data structure, and fix at location in memory

   function IS_EMPTY return BOOLEAN is


       Return TRUE if Start of Data Pointer equals End of Data
         pointer

       otherwise return FALSE

    Specify bodies for routines visible externally
    ==============================================

   procedure RESET is


       Set the start and end pointers to the 1st packet

       Reset the sequence count to zero

   procedure ADD(PCKT : in PACKET.TM_TYPE ) is
```

```
    If the queue is full

       Then raise a TM Q Overflow exception (This should never happen
         as TMQ package should guard against this?)

    Otherwise

       Store packet at next free slot

       Store sequence count in packet

       Prepare sequence count for next packet, performing 'wraparound'
         if necessary

       If CRC required

          Convert packet to an array of 16 bit word

          Calc CRC location in words from pre calc. packet length in bytes

          Calculate CRC value

          and place it at CRC location

       Check here whether queue is now shown as empty.
       If it is then the
       queue was empty prior to packet insertion.
       (Note: this is so because we haven't updated the pointers yet
       and so still reflect pre-insertion status.)

         If so, we need to inform s/c of the new packet address
         (derived from the Output Pointer) which is now available.
         Also tell the spacecraft its length.
         Note that the INPUT_POINTER = OUTPUT_POINTER at this stage.

         Finally, ensure TM_READY bit is up,
         to let spacecraft know about there are packets to take.

       Otherwise

         Do nothing, because there are still packets to be
         removed and therefore the spacecraft has the information
         it needs from a previous pass.

       Finally, calculate next slot index by incrementing the
       input pointer (and 'wrapping around' if necessary).

procedure REMOVE is


    NOTE: This routine should have been removed as its function is now
    performed by a low-level assembler routine in package RBI_IH.

    Ensure TM_READY bit is down while we process this

    Calculate new output index following packet removal

    If the queue is now empty

       Leave TM_READY bit low to inform s/c of the fact

       Otherwise, inform s/c of packet info for next packet fetch

       Ensure TM_READY bit is up, to let s/c more packets to come

function IS_FULL return BOOLEAN is


    Calc index of next (after current) packet slot to be written

    Return TRUE if same as next location to be read

function PACKET_COUNT return UINT16 is


    Return the current packet sequence count
```

#### *6.3.2.57 tmpsu.ads*

```
Extracted from file "tmpsu.ads"


    Function
    ========

    This file contains the specification for the TMPSU package. The package
    contains the software to control and monitor the Telescope Module Power
    Supply. It is based on document XMM-OM/IALS/SP/0002 -
    "TMPSU Electrical Specification".

----------------
package TMPSU is
----------------


   procedure SEND(
                  SUBADR : in  SUB_ADDRESS_TYPE;
                  DATUM  : in  UINT16;
                  OK     : out BOOLEAN);


       Sends the data value DATUM to the MACS subaddress SUBADR of the
       TMSPU. OK is set to TRUE if no errors occur.

   procedure ACQUIRE(SUBADR : in  SUB_ADDRESS_TYPE;
                     DATUM  : out UINT16;
                     OK     : out BOOLEAN);


       Reads the data value DATUM from the MACS subaddress SUBADR of the
       TMSPU. OK is set to TRUE if no errors occur.

   function SET_SECONDARY_VOLTAGES(ON_OFF : BOOLEAN;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN;


       Enables or disables (ON_OFF = TRUE or FALSE respectively)
       the secondary
       voltages that power the blue electronics.
       SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
       associated telecommand.

   function SECONDARY_VOLTAGES_ENABLED return BOOLEAN;


       Returns the status of the Secondary Voltages (TRUE = ON) for display
       in Housekeeping.

   function SET_COARSE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                               SRC_AND_SEQUENCE_COUNT : UINT16)
                                               return BOOLEAN;


       Sets the current for the coarse sensor illuminating LED in 'raw' units
       to be used when moving the filter wheel. The value is not used until
       a call to COARSE_SENSOR is made.
       SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
       associated telecommand.

   function COARSE_SENSOR_CURRENT return UINT16;


       Returns the current for the coarse sensor illuminating LED
       in 'raw' units
       that is used when moving the filter wheel.

   procedure COARSE_SENSOR( ON_OFF : BOOLEAN);


       Turns on/off (ON_OFF = TRUE/FALSE) the illuminating LED used
       by the filter wheel coarse sensor. It uses the current specified in an
       earlier call to SET_COARSE_POSITION_SENSOR_CURRENT.

   function SET_PHASE(DEVICE : in DEVICE_TYPE;
                     PHASE  : in UINT16;
                     SRC_AND_SEQUENCE_COUNT : UINT16)
                     return BOOLEAN;
```

```
    Enables the phase coils for the stepper motor driving DEVICE
    (FILTER_WHEEL or DICHROIC) as specified by the bit pattern contained
    in PHASE (1 = enabled) as follows:

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

function FW_PHASE return UINT16;


    Returns a bit pattern specified by earlier calls to SET_PHASE
    commanding the filter wheel stepper motor for which the bit pattern
    PHASE was non zero. As before, the bits are defined as follows
    ( 1 = enabled)

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------

function DM_PHASE return UINT16;


    Returns the last commanded dichroic phase
    Returns a bit pattern specified by ealier calls to SET_PHASE
    commanding the dichroic stepper motor for which the bit pattern
    PHASE was non zero. As before, the bits are defined as follows
    ( 1 = enabled)

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------

function SET_HEATER_CONFIG(CONFIG : UINT16;
                           SRC_AND_SEQUENCE_COUNT : UINT16)
                                                                    return BOOLEAN;


    The bit pattern in CONFIG specifies which heater should be on or off
    ( 1 = on ) as follows:
                                             L.S.B.
    -------------------------------------------------
    |  Temperature Control  |      Focussing        |
    -------------------------------------------------
    |    Main   |  Forward  | Metering  | Secondary |
    |           |           |   Rods    |  Mirror   |
    |  (HTR 1)  |  (HTR 2)  |  (HTR 3)  |  (HTR 4)  |
    -------------------------------------------------

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

function HEATER_CONFIG return UINT16;


    Returns a bit pattern specifying the current heater configuration
    as follows:
                                             L.S.B.
    -------------------------------------------------
    |  Temperature Control  |      Focussing        |
    -------------------------------------------------
    |    Main   |  Forward  | Metering  | Secondary |
    |           |           |   Rods    |  Mirror   |
    |  (HTR 1)  |  (HTR 2)  |  (HTR 3)  |  (HTR 4)  |
    -------------------------------------------------


function CURRENT(SECONDARY_VOLTAGE : UINT16) return UINT16;


    Returns the current (in 'raw' units ) for the secondary supply circuit
    specified by SECONDARY_VOLTAGE as follows:

    +25  V : 0
    +15  V : 1
    +11  V : 2
```

```
    +5.3 V : 3
    -5.3 V : 4
    -15  V : 5
    +28  V : 6
    + 5  V : 7
```

The values returned are used in the Housekeeping.

```
function COARSE_POSITION_SENSED return BOOLEAN;
```

Returns TRUE if the filter wheel coarse sensor is currently detected.

#### *6.3.2.58 tmpsu.adb*

Extracted from file "tmpsu.adb"

```
 Function
 ========

 This file contains the body for the TMPSU package.  The package
 contains the software to control and monitor the Telescope Module Power
 Supply. It is based on document XMM-OM/IALS/SP/0002 -
 "TMPSU Electrical Specification".

---------------------
package body TMPSU is
---------------------


   procedure SEND(
                  SUBADR : in  SUB_ADDRESS_TYPE;
                  DATUM  : in  UINT16;
                  OK     : out BOOLEAN) is


       Send the DATUM to MACS sub-address SUBADR at the MACS address
       corresponding to the TMPSU on the Instrument Control Bus.

       OK is TRUE if no errors occur.

   procedure ACQUIRE(SUBADR : in  SUB_ADDRESS_TYPE;
                     DATUM  : out UINT16;
                     OK     : out BOOLEAN) is


       Gets the DATUM at MACS sub-address SUBADR at the MACS address
       corresponding to the TMPSU on the Instrument Control Bus.

       OK is TRUE if no errors occur.

   function SET_SECONDARY_VOLTAGES(ON_OFF : BOOLEAN;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN is


       Remember the last commanded secondary status.

       As the bit defining the status of the secondaries is
       combined with other bits, construct the bit pattern from the
       requested status of the secondaries and the last known values
       of the other bits.

       The layout is as follows:
       MSB
       -----------------------------
       |CS0|CS1|CS2|SC0|SC1|SC2|SE |
       -----------------------------

       where CS0->CS2 specify which secondary circuit is being monitored.
             SC0->SC1 specify the coarse sensor illuminating current.
             SE       specifies whether the secondaries are enabled.

       Write the bit pattern to the appropriate address & subaddress
       on the ICB (Macsbus).

       Allow electronics to settle.

       If we had a macsbus error

          Restore record of current status to that of the last status.

       Always return OK as the ICB routines inform the ground if there
       was an error.

   function SECONDARY_VOLTAGES_ENABLED return BOOLEAN is


       Return the recorded status of the secondary supplies.

   function SET_COARSE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                               SRC_AND_SEQUENCE_COUNT : UINT16)
                                               return BOOLEAN is
```

```
    Store the sensor current for later use (note that unlike
    operational mode code there is no check on the value).

    Always return OK.

function COARSE_SENSOR_CURRENT return UINT16 is


    Return the 'raw' current to be used when powering the illuminating
    LED for the filter wheel coarse sensor.

procedure COARSE_SENSOR(ON_OFF : BOOLEAN) is


    If the LED is to be turned on

      Determine the current value from the earlier value( given by
      SET_COARSE_POSITION_SENSOR_CURRENT or a default value).

    otherwise

      specify it as zero

    As the bits defining the 'raw' current to drive the illuminating
    LED of the filter wheel coarse sensor is combined with other bits,
    construct the bit pattern from the determined value of current and
    the last known values of the other bits.

    The layout is as follows:
    MSB
    -----------------------------
    |CS0|CS1|CS2|SC0|SC1|SC2|SE |
    -----------------------------

    where CS0->CS2 specify which secondary circuit is being monitored.
          SC0->SC1 specify the coarse sensor illuminating current.
          SE       specifies whether the secondaries are enabled.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus).

function SET_PHASE(DEVICE : in DEVICE_TYPE;
                   PHASE : in UINT16;
                   SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN is


    It should be noted that the same TMSPU MACSbus sub address
    is used to command the stepper motor phases for both the
    filter wheel and dichroic as follows

      MSB
    -----------------------------------------
    | F1 | F2 | F3 | F4 | D1 | D2 | D3 | D4 |
    -----------------------------------------

    where D1->D4 are the dichroic motor phases.
          F1->F4 are the filter wheel motor phases.

    Determine which device is being commanded.

       If the filter wheel is being commanded

          Insert the requested phase bit pattern into the
          the appropriate part of the command word to be
          to be sent to the mechanisms.

          If it's a non zero phase, remember for recall
          as last active phase for the filter wheel.

       If it's the dichroic that's being commanded

          Insert the requested phase bit pattern into the
          the appropriate part of the command word to be
          to be sent to the mechanisms.

          If it's a non zero phase, remember for recall
          as last active phase for the dichroic.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus).

    Always return OK as the ICB routines inform the ground if there
```

```
    was an error.

function FW_PHASE return UINT16 is


    Return the last non zero phase pattern sent to the filter wheel.

function DM_PHASE return UINT16 is


    Return the last non zero phase pattern sent to the dichroic.

function SET_HEATER_CONFIG(CONFIG : UINT16;
                           SRC_AND_SEQUENCE_COUNT : UINT16)
                           return BOOLEAN is


    Loop over permitted heater configurations

       If the request heater configuration is one of them

         Write the bit pattern to the appropriate address & subaddress
         on the ICB (Macsbus).

          Remember the requested heater configuration for
          HK and heater control purposes.

          and exit with a success flag.

    Otherwise exit (in this release, also with a success flag).

function HEATER_CONFIG return UINT16 is


    Return the last commanded heater configuration.

function CURRENT(SECONDARY_VOLTAGE : UINT16) return UINT16 is


    If the requested circuit is outside the allowed range of circuits

       return 0

    As the bits defining which secondary circuit is to be monitored are
    combined with other bits, construct the bit pattern from the
    requested secondary circuit and the last known values
    of the other bits.

    The layout is as follows:
    MSB
    -----------------------------
    |CS0|CS1|CS2|SC0|SC1|SC2|SE |
    -----------------------------

    where CS0->CS2 specify which secondary circuit is being monitored.
          SC0->SC1 specify the coarse sensor illuminating current.
          SE       specifies whether the secondaries are enabled.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus).

    Wait for electronics to settle.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus) to initiate an analogue to digital conversion.

    Wait a bit

    Get datum containing the value from the appropriate address
    on the MACSbus.

    The format of the datum now received is as follows:

    -------------------------------------------------
    |C0|C1|C2|C3|C4|C5|C6|C7|XX|XX|XX|XX|XX|XX|XX|CS|
    -------------------------------------------------

    where C0->C7 is the 'raw' current of the requested secondary circuit.
          XX is "don't care".
          CS is coarse sensor status, 1 = 'seen'


    Extract current value from the C0->C7 field within the datum
```

```
    and return it.

function COARSE_POSITION_SENSED return BOOLEAN is


    Get datum containing the value from the appropriate address
    on the MACSbus.

    The format of the datum now received is as follows:

    -------------------------------------------------
    |C0|C1|C2|C3|C4|C5|C6|C7|XX|XX|XX|XX|XX|XX|XX|CS|
    -------------------------------------------------

    where C0->C7 is the 'raw' current of the requested secondary circuit.
          XX is "don't care".
          CS is coarse sensor status, 1 = 'seen'.


    Extract sensor status from the CS field within the datum
    and return it.
```

#### 6.3.2.59 tmq.ads

Extracted from file "tmq.ads"

```
    Function

    ========

    This file contains the specification for the TMQ package.
    The function of that package is to provide routines to control
    access to the telemetry queue

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

    The protocol it implements is defined in the OBDH Bus Protocol
    Requirement Specification XM-IF-DOR-0002

--------------
package TMQ is
   --------------


   procedure RESET;


      The procedure RESET resets (i.e. clears) the telecommand queue

   procedure REMOVE;


      The procedure REMOVE is called upon receipt of an EOTM Instruction to
      User from the spacecraft. This indicates that a TM packet has been
      taken

      NOTE: This routine should be removed as its function is now
      performed by a low-level assembler routine in package RBI_IH.

   task GUARDED is
      pragma PRIORITY(IMPORTANCE.TMQ_GUARDED);
      entry PUT(PCK : in PACKET.TM_TYPE);
   end GUARDED;


         PUT access to the telemetry queue is via the above task GUARDED
         to force queuing for access to the TM queue.

         The task entry PUT places a packet in the telemetry queue

           where:

           LEVEL   indicates the priority

           PCK     is the packet to be inserted into the queue.


   function PACKET_COUNT return UINT16
     renames TM_Q.PACKET_COUNT;


      Rename (for convenience) the PACKET_COUNT function of package TM_Q.
```

#### *6.3.2.60 tmq.adb*

Extracted from file "tmq.adb"

```
 Function
 ========

 This file contains the body for the TMQ package.
 The function of that package is to provide routines to control
 access to the telemetry queue. It, in turn, call lower level routine in
 package TM_Q.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

 The protocol it implements is defined in the OBDH Bus Protocol
 Requirement Specification XM-IF-DOR-0002

-------------------
package body TMQ is
-------------------


    where:

    PCK is the packet to be inserted into the queue

   procedure SEND_TO_TM_Q (PCK : in PACKET.TM_TYPE) is


       Commence infinite loop

          If the telemetry queue is full

             Wait a bit

          Otherwise

             Place packet in queue (via TM_Q.ADD)

             and exit from loop

          end infinite loop

   task body GUARDED is


       First, reset the telemetry queue.

       Then commence infinite loop

          Now wait on a rendevous at the PUT entry point

                Send the packet to the telemetry queue
                (via SEND_TO_TM_Q)

          End of infinite loop

   procedure RESET is


       Reset the telemetry queue

   procedure REMOVE is


     Call the 'remove packet' routine for the telemetry queue.

     NOTE: This routine should have been removed as its function is now
     performed by a low-level assembler routine.
```

#### 6.3.2.61 types.ads

Extracted from file "types.ads"

```
Function
========

The function of this package specification is to define the basic data
types used throughout the ICU ADA code.

Definitions
===========

   Define Unsigned Byte type UBYTE

   Define Signed Byte type BYTE

   Define Unsigned 16 bit integer type UINT16

   Define Signed 16 bit integer type INT16

   Define Signed 32 bit type INT32

   Define Unsigned Byte Unconstrained Array type UBYTE_ARRAY

   Define Signed Byte Unconstrained Array type BYTE_ARRAY

   Define Unsigned 16 bit Integer Unconstrained Array type UINT16_ARRAY

   Define Signed 16 Bit Integer Unconstrained Array type INT16_ARRAY

   Define Unsigned Nibble type

   Define Unsigned Nibble Array Type

   Define single bit Integer Unconstrained Array type BIT_ARRAY
```

#### ***6.3.2.62 USERDEFS.asm***

**File is USERDEFS.asm**

## 6.4 Operational Code

Operational code is built from the following files:-

| ADA | | Assembler |
|---|---|---|
| **Specifications** | **Bodies** | |
| bcp4_ih.ads | | bcp4_ih.asm |
| crc.ads | crc.adb | |
| debug.ads | debug.adb | |
| dempsu.ads | dempsu.adb | |
| detanalog.ads | detanalog.adb | |
| detdigital.ads | detdigital.adb | |
| detector.ads | | |
| dpu.ads | dpu.adb | |
| dpu_mem_manager.ads | dpu_mem_manager.adb | |
| dpu_mnemo.ads | | |
| heater.ads | heater.adb | |
| hk.ads | hk.adb | |
| icb.ads | icb.adb | |
| icb_driver.ads | icb_driver.adb | |
| | icu.ada | |
| icu_mem_manager.ads | icu_mem_manager.adb | |
| | | INTVEC.ASM |
| importance.ads | | |
| mechanism.ads | mechanism.adb | |
| mem_manager.ads | mem_manager.adb | |
| memdpu.ads | memdpu.adb | |
| memloc.ads | | |
| modeman.ads | modeman.adb | |
| mutex.ads | mutex.adb | |
| nhk.ads | nhk.adb | |
| packet.ads | | |
| peek_poke.ads | | peek_poke.asm |
| rbi.ads | rbi.adb | |
| rbi_ih.ads | | rbi_ih.asm |
| reset.ads | | reset.asm |
| science_fm.ads | science_fm.adb | |
| ssi_driver.ads | ssi_driver.adb | |
| ssi_ih.ads | | ssi_ih.asm |
| ssi_in.ads | ssi_in.adb | |
| ssi_out.ads | ssi_out.adb | |
| task_report.ads | task_report.adb | |
| taskman.ads | taskman.adb | |
| tc_q.ads | tc_q.adb | |
| tc_verify.ads | tc_verify.adb | |
| tcq.ads | tcq.adb | |
| time_man.ads | time_man.adb | |
| timer_a_ih.ads | timer_a_ih.adb | |
| tm_man.ads | tm_man.adb | |
| tm_q.ads | tm_q.adb | |
| tmpsu.ads | tmpsu.adb | |
| tmq.ads | tmq.adb | |
| types.ads | | |
| | | USERDEFS.ASM |

The following pages contains 'Structured English' extracted from comments in the file. They should be studied in conjunction with the code listings as they have additional comments regarding implementation details but are omitted in this document for clarity.

- The comments extracted from the specification files (*.ads) describe **'what'** a given package does.
- The comments extracted from the associated body files (*.ads or *.asm) describe **'how'** a given package performs the operations defined by the specification.

In addition, the file `icu.xtof` can be supplied. It may be used in conjunction with the TARTAN utility `adaref1750a` to extract the dependencies, list of calls and inverse calls and cross reference information.

To extract the call graph (of ‘callers’).

```
adaref1750a –input icu.xtof –call_graph
```

To extract the call graph (of ‘called by’).

```
adaref1750a –input icu.xtof –call_graph –reverse
```

To extract the call graph (of ‘callers’) from one package.

```
adaref1750a –input icu.xtof –call_graph –from package_name
```

To extract a list of dependent relationships.

```
adaref1750a –input icu.xtof –dependency_graph
```

To extract a list of dependent relationships from one package.

```
adaref1750a –input icu.xtof –dependency_graph –from package_name
```

To extract a alphabetical list of user defined entities, containing source location of declaration, source location of where it is set and used.

```
adaref1750a –input icu.xtof –xref
```

To extract a alphabetical list of user defined entities, containing source location of declaration, source location of where it is set and used for one package.

```
adaref1750a –input icu.xtof –xref –about package_name
```

## 6.4.1 Main Program

### *6.4.1.1 icu.ada*

Extracted from file "icu.ada"

```
 Function
 ========

 This procedure is the 'main' program for the ICU. It

 1) Initialises the ICU then...
 2) Routes all valid received telecommand packets as appropriate

procedure ICU is


      Initializations
      ===============


      Initialise RBI related matters
      (including the communications area and TC and TM ready bits)

      Start the RBI Watchdog.

      Reset the ICB interface

      Wait a bit, then turn on secondary power,
      thus enabling the blue electronics.

      Once secondaries settled, we now initialise the mechanism
      package (primarily to ensure we have an initial value of the
      coarse and fine sensors to be used in housekeeping)

      First ensure actual initial configuration is the same as default
      assumed in code.

      Then start the automatic heater algorithms

      Then start the automatic heater control algorithms

      Ensure that telemetry queues are initialised

      Ensure the telecommand queues are initialised (after which we can
      receive telecommands

      Now start the DPU processing package.

      Now start the Housekeeping.

      Now begin the endless control loop

        Wait for a valid telecommand packet (via TMQ.GET procedure)

        When a valid packet is obtained, route it to the appropriate package
        on the basis of the packet type

          For a Task Management Packet

            Send it to the TASKMAN package

          For a Memory Maintenance Packet

            Send it to the MEM_MANAGER package.

          For a Telemetry Management Packet

            Send it to the TM_MAN package.

          For a Time Management Packet

            Send it to the TIME_MAN package.

          For a Test packet

            do nothing

          For all other packet types
```

```
    do nothing (as this shouldn't happen)

end of selection by packet type

If nothing has indicated that the packet was bad

  Place a Successful Acceptance Telemetry Packet in the
  telemetry queue.

  Increment the 'Good Packet' counter (modulo 65536) for
  inclusion in the HK.

Otherwise

  Increment the 'Bad Packet' counter (modulo 65536) for
  inclusion in the HK.

End the controlling loop
```

### 6.4.2 Packages

#### *6.4.2.1 bpc4_ih.ads*

**Extracted from file "bcp4_ih.ads"**

```
    Function
    ========

    This file merely contains the specification for the XMM-OM bcp4 interrupt
    handler.  It specifies that the body of bcp4_ih is written in assembler
    and therefore directs the linker to link it as foreign.
    The interrupt handler had to be written in assembler for speed so as not to
    block other interrupts for too long.

package BCP4_IH is
   pragma FOREIGN_BODY("ASM");
end BCP4_IH;
```

#### *6.4.2.2 bcp4.ih.asm*

**File is bcp4_ih.asm**

```
     Save some space for the Linkage Pointer
        Save some space for the Stack Pointer
        Registers r0-r1 can be trashed.
        All other registers must be preserved.
        So save R0 and R1 on the stack
    Fetch the interrupt counter
    Check for impending overflow (is it 7fffh)
    If it's OK (not 7fffh), increment it
    otherwise avoid an overflow by setting it to 8000h
    Then write it back to memory
    Check the BCP flag and if it is not 1, we don't have to bother doing any
    work so jump to to the cleanup and end
    "Freeze" the current time by reading the "freeze_obt_instr" register
    and writing the value to the config register.
    Perform dummy xio to give the RBI time to freeze (just a delay)
    Read bits 0-15
    and write to memory
    Read bits 16-31
    and write to memory
    Read remaining bits 32-42 (result in high order bits)
    and write to memory
    Set the BCP flag (in memory) to 2 to show we've now got a time available
    Recover registers
    Turn interrupts back on
    Return back from whence we came
```

#### *6.4.2.3 crc.ads*

Extracted from file "crc.ads"

```
    Function
    ========
    This file contains the specification for the CRC package.
    This contains the CRC algorithms for XMM which
    are based on the algorithm described in ESA technical note PX-TN-00540

--------------
package CRC is
--------------


  function CALC(DATA : UBYTE_ARRAY; NUMBER : UINT16 ) return UINT16;


      This function returns the unsigned 16 bit integer checksum of the
      first NUMBER locations in unsigned byte array DATA.

  function CHECK_TC(TC : PACKET.TC_TYPE) return UINT16;


      This function calculates the checksum of a whole TC packet,
      using the packet length stored within the packet to determine its
      length. Returns value of zero if as expected, otherwise returns
      value of checksum found, NOT including the 2 byte checksum
      field at the end of the packet.
      It thus checks whether that packet TC contained a valid CRC.

  function CALC_TM(TM : PACKET.TM_TYPE) return UINT16;


      This function calculates the value to be inserted into
      the checksum field of packet TM, using the packet length stored
      within the packet to determine the length of the data to be checksumed
      (i.e. NOT including the checksum field at the end of the packet).

  function CALC_MEM(CURRENT_CRC : UINT16;
                    MEM : UINT16_ARRAY;
                    NO_WORDS : INTEGER) return UINT16;


      This function is used to calculate a checksum for a large block
      of data on the assumption that not all the data will be available
      at once. Therefore, it uses the CURRENT_CRC value returned by a prior
      call as input to the current call and then calculates the CRC of the
      NO_WORDS 16-bit words of data contained in MEM. The result is the CRC
      for all blocks of data supplied (NOTE: the sequence is restarted by
      supplying a value of all binary ones for CURRENT_CRC).
```

#### *6.4.2.4 crc.adb*

Extracted from file "crc.adb"

```
 Function
 ========
 This file contains the body for the CRC package.
 This contains the CRC algorithms for XMM which
 are based on the algorithm described in ESA technical note PX-TN-00540

-------------------
package body CRC is
-------------------


  function CLC(SYNDROME : UINT16; DATA : UBYTE_ARRAY;  NUMBER : UINT16 )
    return UINT16 is


     This function returns the unsigned 16 bit integer checksum of the
     first NUMBER locations in unsigned byte array DATA. An initial value
     of the currently 'running' checksum is contained in SYNDROME.
     It is a function internal to this package.

     The following test data was used (taken from the reference above).

          DATA                CRC
          ++++                +++
     00 00                   1D 0F
     00 00 00                CC 9C
     AB CD EF 01             04 A2
     14 56 F8 9A 00 01       7F D5

   First define the lookup table for efficient calculation (equivalent of
   routine InitLtbl in above reference.

     loop over NUMBER data points

       Calculate RHS term by

       1) Shift right the input checksum by 8.

       2) Exclusive Or result with current datum.

       3) Mask off the 8 least significant bits of the result.

       4) Use result to index into table of pre-calculated coefficients.

       Calculate LHS term by

       1) Shift left the input checksum by 8.

       2) Mask off the 8 most significant bits of the result.

       Calculate checksum by Exclusive Oring the two terms.

     Return final value of the checksum.

  function CALC(DATA : UBYTE_ARRAY;  NUMBER : UINT16 ) return UINT16 is


     Call the CLC routine with SYNDROME set to all binary 1's.

  function CHECK_TC(TC : PACKET.TC_TYPE) return UINT16 is


     This function calculates the checksum of a whole packet,
     using the packet length stored within the packet to determine its
     length. Returns value of zero if OK, otherwise returns
     value of checksum found, NOT including the 2 byte checksum
     field at the end of the packet.
     It thus checks whether that packet contained a valid CRC.

     Call routine CALC (using the whole packet as data and deriving
     its length from internal length information) to check that the result
     (i.e. the checksum of whole packet) is zero

       If it is, return zero

     Otherwise
```

```
     Return checksum found (not including the CRC field).

function CALC_TM(TM : PACKET.TM_TYPE) return UINT16 is


   This function calculates the value to be inserted into
   the checksum field of packet TM, using the packet length stored
   within the packet to determine the length of the data to be checksumed
   (i.e. NOT including the checksum field at the end of the packet).

    Calculate the appropriate length to be used from the length
    field in the packet, then use routine CALC to calculate the
    checksum of packet TM and return the value.

function CALC_MEM(CURRENT_CRC : UINT16;
                  MEM : UINT16_ARRAY;
                  NO_WORDS : INTEGER) return UINT16 is


   This function is used to calculate a checksum for a large block
   of data on the assumption that not all the data will be available
   at once. Therefore, it uses the CRC value returned by a prior
   call as input to the next one.

   Loop over the block of data, 1 16 bit word at a time.

     Call function CLC to calculate the 'running' CRC for just 1 word.

   Return the resulting CRC.
```

#### *6.4.2.5 debug.ads*

Extracted from file "debug.ads"

```
    Function
    ========

    This file contains the specification and body for the package DEBUG.
    As its name implies, it contains a collection of routines useful
    for debugging.

    Dependencies
    ============

with TYPES; use TYPES;
with SYSTEM;
with MEMLOC;


----------------
package DEBUG is
   ----------------


   ----------------------------------
   procedure PROGRESS(ITEM : UINT16);
   ----------------------------------


       Where ITEM is the progress number to write to memory

   ------------------------------------------
   procedure PROGRESS_SPECIAL(ITEM : UINT16);
   ------------------------------------------


       Where ITEM is the progress number to write to memory
       This procedure writes the number "ITEM" to a fixed location in memory
       and is used to keep a record of how far the running code has progressed.
       When this memory location is read later, after a crash, it will provide
       good idea as to what was running as the code crashed.

   -------------------------------------------
   procedure PROGRESS_SPECIAL2(ITEM : UINT16);
   -------------------------------------------


       Where ITEM is the progress number to write to memory
       This is another progress indicator like the above.

   ------------------------------------------
   procedure EXCEPTION_REPORT(ITEM : UINT16);
   ------------------------------------------


       Where ITEM is the exception number to write to memory
       When the running code produces an Ada exception, the Ada exception
       handler should call this procedure which will write the exception
       number to a special known location in memory that can be read afterwards
       to help understand why the code crashed.

       Define some constants for the progress and exception numbers.
       In this way, the high order bits of the code numbers used indicate the
       package involved. These are detailed in the introduction to the
       Detailed Design Document.
```

#### *6.4.2.6 debug.adb*

Extracted from file "debug.adb"

```
 Function
 ========

 This file contains the specification and body for the package DEBUG.
 As its name implies, it contains a collection of routines useful
 for debugging.

with PACKET;
with NHK;


package body DEBUG is


   procedure PROGRESS(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory

       ITEM identifies which part of the code is running: the package and
       a location in that package

       If we haven't had an Ada exception

          Write ITEM to the FIRST_EXCEPTION standard memory location
          ITEM identifies which part of the code is running: the package and
          a location in that package
          After an Ada exception the value stored at this address
          will not change

       Write ITEM to the LAST_PROGRESS memory location
       This will continue to update after an Ada exception

   procedure PROGRESS_SPECIAL(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory
       Like procedure package, this writes a vaule to a special location
       in memory for debug purposes.  It is used so as not to interfere
       with the location used by procedure PROGRESS.

       Write ITEM to a standard memory location
       (also called PROGRESS_SPECIAL)

   procedure PROGRESS_SPECIAL2(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory

       Write ITEM to a standard memory location
       (also called PROGRESS_SPECIAL2)

   procedure EXCEPTION_REPORT(ITEM : UINT16) is


       Where ITEM is the progress number to write to memory

       If this is the first exception trapped

          Write ITEM to the fixed memory location FIRST_EXCEPTION
          reserved to store the first exception.
          This will not be overwritten.
          ITEM identifies in which part of the code the exception occured:
          the package and which exception was handled

       Then write ITEM to the fixed memory location reserved to store the
       last exception (LAST_EXCEPTION).
       This is overwritten at each exception.
```

#### *6.4.2.7 dempsu.ads*

Extracted from file "dempsu.ads"

```
    Function
    ========

    This file contains the specification for the DEMPSU package
    It provides routines to control the Digital Electronics Module
    Power Supply Unit.

-----------------
package DEMPSU is
-----------------


  procedure DPU_RESET;


      Resets the DPU after a 'latch-up' or turns it on again if it is
      powered down.
```

#### *6.4.2.8 dempsu.adb*

Extracted from file "dempsu.adb"

```
 Function
 ========

 This file contains the body for package DEMPSU
 It provides routines to control the Digital Electronics Module
 Power Supply Unit.

----------------------
package body DEMPSU is
----------------------


  procedure DPU_RESET is


     To reset/turn on the DPU, write a "don't care" bit
     pattern to the DPU Reset Register of the DEMPSU control card.
```

#### *6.4.2.9 detanalog.ads*

Extracted from file "detanalog.ads"

```
    Function
    ========

    This file contains the specification for the detanalog package. It
    controls the analogue card of the detector electronics. This card is
    described in document XMM-OM/MSSL/SP/81.2, "Blue Detector Analogue Card
    Requirement Specification"

---------------------
package DETANALOG is
---------------------


  function SET_FINE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                            SRC_AND_SEQUENCE_COUNT : UINT16)
                                            return BOOLEAN;


      CURRENT specifies the illuminating LED current (in 'raw' units)
      to be used for the filter wheel fine position sensor when the filter
      wheel is moved.

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.

  function FINE_SENSOR_CURRENT return UINT16;


      Returns the current (in 'raw' units) for the fine sensor
      specified by an earlier call to SET_FINE_POSITION_SENSOR_CURRENT.

  procedure FINE_SENSOR(ON_OFF : BOOLEAN);


      If ON_OFF is TRUE, turns on the illuminating LED of the Filter Wheel Fine
      Sensor using a 'raw' current value supplied by an earlier call to
      SET_FINE_POSITION_SENSOR_CURRENT.
      If ON_OFF is FALSE, the current is set to zero.

  function FINE_POSITION_SENSED return BOOLEAN
    renames TIMER_A_IH.FINE_POSITION_SENSED;


      Returns TRUE when the filter wheel fine position sensor is detected

  function  SET_FLOOD_LED_BIAS_CURRENT(LED : UINT16;
                                       SRC_AND_SEQUENCE_COUNT : UINT16)
                                       return BOOLEAN;


      Sets the flood led's bias current to the value in LED ('raw' units).

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.

  procedure SET_HV_ENABLE(ENABLED : BOOLEAN);


      Enable or Disables (ENABLED = TRUE or FALSE respectively) the High Voltage
      Facility on the analogue card.

      NOTE: This is done by writing to the appropriate ICB MACSbus port
      with the relevant bit set.
      It should be noted that this port is also used to set the value for
      Vmcp1. Consequently, the last value of Vmcp1 requested is resent.

  procedure SET_HV(HV        : HV_TYPE;
                   VALUE     : UINT16);


      Sets the High Voltage HV to 'raw' bit pattern VALUE.
      The raw bit pattern is obtained from CONVERT_HV_TO_BITS.
```

```
    HV specifies one of mcp23 (DETECTOR.V_MCP23)
                        mcp1 (DETECTOR.V_MCP1) or
                        Vcathode (DETECTOR.V_CATHODE,

function LOAD_HV_RAMP_PARAMETERS(VOLTAGE : UINT16;
                                 VALUE : UINT16;
                                 RAMP_RATE : UINT16;
                                 FORCE : UINT16;
                                 SRC_AND_SEQUENCE_COUNT : UINT16)
                                 return BOOLEAN;


    Loads and checks the ramp parameters for a single mcp voltage

    where :

    VOLTAGE   specifies one of mcp23, mcp1 or Vcathode
    VALUE     is the voltage level required
    RAMP_RATE is the rate of ramping in volts/second
    FORCE     causes the hv ramp task to ignore errors
    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
              of the associated telecommand.

    Returns TRUE if the command was successfully accepted


function HV_RAMP_START(SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN;


    Starts the hv ramp task

    where :

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if the command was successfully accepted


function HV_RAMP_STOP(SRC_AND_SEQUENCE_COUNT : UINT16)
                      return BOOLEAN;


    Stops the hv ramp task

    where :

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if the command was successfully accepted


function PERFORM_HV_SAFING(LEVEL : UINT16; SRC_SEQ_COUNT : UINT16) return BOOLEAN;


    Performs safing of the high voltages

    where :

    LEVEL determines whether to perform full (DETECTOR.FULL_SAFE)
    or intermediate safing (DETECTOR.HALF_SAFE).
    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if the function was successfully executed


function SAFE_ONE_HV(VOLTAGE : HV_TYPE; SRC_SEQ_COUNT : UINT16) return BOOLEAN;


    Safes one high voltage

    where :

    VOLTAGE specifies one of mcp23 (DETECTOR.V_MCP23)
                             mcp1 (DETECTOR.V_MCP1) or
                             Vcathode (DETECTOR.V_CATHODE,

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.
```

```
    Returns TRUE if the function was successfully executed

function GET_SET_GO(VOLTAGE : HV_TYPE) return INTEGER;

    Checks that value is between the upper and lower ranges (obsolete)

function CONVERT_HV_TO_BITS(VOLTAGE : HV_TYPE; VALUE : INTEGER) return UINT16;

    Converts the value of voltage to a bit pattern suitable for output to the hv card

    where :

    VOLTAGE specifies one of mcp23 (DETECTOR.V_MCP23)
                             mcp1 (DETECTOR.V_MCP1) or
                             Vcathode (DETECTOR.V_CATHODE,

    VALUE is the voltage level requested in engineering units (volts)

    Returns UINT16 bit pattern representing VALUE

function GET_CONVERTED_HV(VOLTAGE : HV_TYPE) return INTEGER;

    Gets the hv level of voltage in engineering units

    where :

    VOLTAGE specifies one of mcp23 (DETECTOR.V_MCP23)
                             mcp1 (DETECTOR.V_MCP1) or
                             Vcathode (DETECTOR.V_CATHODE,

    Returns the voltage level


procedure SET_ADC_ACCURACY(ACCURACY : UINT16);

    Sets the analogue to digital accuracy of the card as follows:
       5 = 1 %
       7 = 0.1 %
       9 = 0.01%

function  GET_ADC_ACCURACY return UINT16;

    Gets the analogue to digital accuracy of the card as specified by SET_ADC_ACCURACY

function GET(ADC_ITEM : UINT16)
             return UINT16;

    Initiates an analogue to digital conversion of channel ADC_ITEM to collect and
    returns its value measured to accuracy ACCURACY set by SET_ADC_ACCURACY.
    The items are as follows:

    Channel     Description
    -------     -----------
       0        Thermistor 0 - BPE
       1        Thermistor 1 - Reference B
       2        Thermistor 2 - Reference C
       3        Thermistor 3 - Main
       4        Thermistor 4 - Forward 1
       5        Thermistor 5 - Forward 2
       6        Thermistor 6 - CCD
       7        Thermistor 7 - Reference A
       8        Vcathode
       9        Vmcp1
      10        Vmcp23
      11        +5V
      12        +15V
      13        -15V
      14        Precision Reference Voltage
      15        Filter Wheel Analogue Reference

    Note - due to noise 'spikes' on the returned values, 5 readings are taken
    in quick succession and an average of the middle 3 in value is returned.

function HV_ENABLED return BOOLEAN;
```

Returns the status of the HV enabled flag from the status word.

```
function FLOOD_LED_BIAS_CURRENT
         return  UNIBBLE ;
```

Returns the value of the last commanded flood led current

#### *6.4.2.10 detanalog.adb*

```
Extracted from file "detanalog.adb"


 Function
 ========

 This file contains the body for the detanalog package. It
 controls the analogue card of the detector electronics. This card is
 described in document XMM-OM/MSSL/SP/81.2, "Blue Detector Analogue Card
 Requirement Specification". This defines the data structures used in this
 package.

-------------------------
package body DETANALOG is
-------------------------


  function SET_FLOOD_LED_BIAS_CURRENT(LED : UINT16;
                                      SRC_AND_SEQUENCE_COUNT : UINT16)
                                      return BOOLEAN is


     If the ICU is not in engineering mode

       Send ground an appropriate command rejection message

       And return a failure condition of FALSE.

     Update the record of data about to be written to port
     and ensure it is within range (max value 15).
     Note - as the flood LED port is also used to control the Fine Sensor LED
     Current for the filter wheel, we must merge the supplied flood LED bit
     pattern with last recorded value used to command the fine sensor LED.

     Write result to appropriate port on the ICB MACSbus.

     Return a success condition of TRUE.
     Note, in the event of a ICB error at this point,
     the ground should notice that the ICB error
     count has increased.

  procedure SET_HV_ENABLE(ENABLED : BOOLEAN) is


     Update the data to be written to port (this is a merging of the
     requested HV enable setting and the last Vmcp1 commanded (as they
     share the same port).
     (Note, no failsafe commanding of voltage levels, we assume
     user knows what they're doing!)

     Write result to the port on the ICB MACSbus.

  procedure SET_HV(HV        : HV_TYPE;
                   VALUE     : UINT16) is


     Examine which HV is being commanded.

       If it is Vcathode

         Merge with previous value used for Vmcp23
         (as they share the same port)

         Make a note we are to write to that port

       If it is Vmcp1

         Merge with previous value used for HV enable
         (as they share the same port)

         Write result to the port on the ICB MACSbus.

       If it is Vmcp23

         Merge with previous value used for Vcathode
         (as they share the same port)

         Make a note we are to write to that port

     If we noted that we are to write to the Vcathode/Vmcp23 port
```

```
     Write the merged values to that port on the ICB MACSbus.

function GET(ADC_ITEM : UINT16)
             return UINT16 is


   Ensure exclusive use of the MUX channel using a mutex semaphore
   (this is to prevent other routines
   selecting another channel while we are still processing this one).

   Specify required MUX Channel by writing the channel number
   to the appropriate 'Set MUX Address' ICB MACSbus port.

   Allow analogue voltage to settle.

   Repeat the following 5 times.

     Start ADC Conversion by reading from the 'Start ADC' port.

     Wait 10 ms

     Read from the 'ADC Read' port

     Extract data bit field from returned datum and store it in a table.

   Sort into order the 5 returned values.

   Release MUX channel for use by others by clearing MUTEX semaphore.

   Return average of the middle 3 of the sorted values.

function HV_ENABLED return BOOLEAN is


    Get Datum from the appropriate MUX port on the ICB MACSbus.

    Extract the bit from the datum corresponding to the HV Enabled status
    and return it.

procedure SET_ADC_ACCURACY(ACCURACY : UINT16) is


   Note requested accuracy in variable ADC_ACCURACY.

function  GET_ADC_ACCURACY return UINT16 is


   Return requested accuracy stored in variable ADC_ACCURACY.

function FLOOD_LED_BIAS_CURRENT
         return  UNIBBLE is


   Return the value of the last flood LED value written.

function SET_FINE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                          SRC_AND_SEQUENCE_COUNT : UINT16)
                                          return BOOLEAN is


    Ensure that value supplied does not exceed maximum,
    then store its value in variable SENSOR_CURRENT,
    but perform no other action.

    Always return success (TRUE).

function FINE_SENSOR_CURRENT return UINT16 is


    Return the last value of the Fine Sensor current supplied
    to SET_FINE_POSITION_SENSOR_CURRENT stored in variable SENSOR_CURRENT.

procedure FINE_SENSOR(ON_OFF : BOOLEAN) is


    If the sensor is to be turned on

      Construct the datum to be used to write to the appropriate port
      using the last supplied value of Fine Sensor current stored in
      SENSOR_CURRENT with the last recorded Flood LED current
      (this is because it shares the port with the Flood LED control port).
```

```
   Otherwise

      Construct the datum to be used to write to the appropriate port
      using a zero value for Fine Sensor current
      and the last recorded Flood LED current
      (this is because it shares the port with the Flood LED control port).

   Write the datum to the appropriate port on the ICB MACSbus.

function LOAD_HV_RAMP_PARAMETERS(VOLTAGE                 : UINT16;
                                 VALUE                   : UINT16;
                                 RAMP_RATE               : UINT16;
                                 FORCE                   : UINT16;
                                 SRC_AND_SEQUENCE_COUNT  : UINT16)
                                 return BOOLEAN is


       then send 'busy' error report and return false

    If the requested mcp voltage is in range

       then check rest of the tc parameters

          If requested voltage is Vcathode

             Cathode must be less than or equal to Vmcp1 or zero

         If requested voltage is Vmcp1

            Vmcp1 must be for turn on :
                    below V mcp23
                    greater than V cathode
            V mcp23 must be greater than the mcp23_lower_limit for mcp1 to rise

            For turn off, V cathode must already be off

         If requested voltage is Vmcp23

            If not turning off, mcp23 must be
                    greater than mcp1
                    gretaer than the mcp1 collapse voltage if mcp1 is on
            If turning off, both mcp1 and cathode must already be off

       Also check ramp rate is valid and that the FORCE parameter is valid.

    If parameters check OK

       then save copy of parameters in a table for later use
       by HV_RAMP_START and HV_RAMP_STOP.

       and return success (TRUE)

    else error in parameters

       so mark all parameters as undefined

       and send an illegal parameters values error packet

       and return a failure condition (FALSE).

function HV_RAMP_START(SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN is


    If in safe mode

       then send 'invalid mode' error report and return false

    If already ramping, we cannot start another ramp

       then send 'busy' error report and return false

    If the HV ramp parameters are not already defined

       then send 'parameters not loaded' error report and return false

    All seems to be in order, lets hope it's not going to blow the instrument up
    Start the HV ramping task by calling HV_PROCESS entry START
    and return true

function HV_RAMP_STOP(SRC_AND_SEQUENCE_COUNT : UINT16)
                      return BOOLEAN is
```

```
    Pass the stop message onto ramp task HV_PROCESS by calling entry STOP.
task body HV_PROCESS is

    Start infinite loop

       Await...

          1) a call to  START entry point

             Copy current settings so that load task won't interfere

             Determine direction of ramp and
             set up controlling parameters accordingly

             But if we are ramping up and filter wheel is not in blocked

                Issue appropriate execution failed message

                and return without setting task running flag.

             Otherwise

                Initialize current value to previous level

                If Vmcp23 is not off then ensure HV enabled bit is set

                Initialize variables for actual HV task

                Set task running flag

          2) a call to the STOP entry point

             Set running flag to false

             Record voltage attained before being stopped

             Set HV parameters to undefined

             Send unsuccessful execution packet to ground

          Otherwise, if task is enabled to run

            wait a bit

             Loop 10 times

                Get current voltage setting

                Set ramped ok flag if voltage in range

                Force exit from loop if ramped ok flag set

                Force exit from loop if call to STOP entry received

                      Set the STOPPED flag if forced exit.

                or

                   wait a second.

             If voltage level is OK

                   Calculate next voltage level

                Perform range check and adjust if necessary

                   Convert voltage level to bits using CONVERT_HV_TO_BITS

                   Output bits

             else either error in ramping or voltage level reached

                Set up error codes

                Ensure HV parameters are not defined and stop task

                If not ramped ok
```

```
                  Send unsuccessfull execution packet

               else either ramp stopped or completed successfully

                  If ramp stopped

                     Send unsuccessfull execution packet

                  Else ramped ok

                     If turning off Vmcp23

                        Then disable HV

                     Send blue event report ...

                     ... only if source sequence count is not FFFF
                     because internal commands have a source and sequence
                     count of this value

function CONVERT_HV_TO_BITS(VOLTAGE : HV_TYPE;
                            VALUE   : INTEGER)
                            return UINT16 is


    If voltage level in the lower band then

       Convert convert voltage

       If channel is primary then

          Perform rounding and adjust if necessary

    Else voltage level is in higher band
    Convert to bit pattern

       If channel is primary

          Perform rounding and adjust if necessary

    Return bit pattern

function GET_CONVERTED_HV(VOLTAGE : HV_TYPE) return INTEGER is


    Return the value read in from the adc, converted to a voltage level

function GET_SET_GO(VOLTAGE : HV_TYPE) return INTEGER is


    Now obsolete

function PERFORM_HV_SAFING(LEVEL : UINT16;
                           SRC_SEQ_COUNT : UINT16)
                           return BOOLEAN is


    If the safing level is a full safe

       then loop through voltages setting them to zero
       using SAFE_ONE_HV

          Return false if safing failed

    else safe only the cathode with SAFE_ONE_HV

function SAFE_ONE_HV(VOLTAGE : HV_TYPE;
                     SRC_SEQ_COUNT : UINT16)
                     return BOOLEAN is


    If hv is not enabled then do nothing and return true

    Set up the ramp parameters with appropriate values
    using LOAD_HV_RAMP_PARAMETERS

       Start the hv ramp task HV_RAMP_START

          Wait for the hv ramp task to finish

          If voltage ramped OK return true else return false
```

```
Else if we couldn't load the ramp parameters

   return FALSE.
```

#### *6.4.2.11 detdigital.ads*

Extracted from file "detdigital.ads"

```
    Function
    ========

    This file contains the specification for the detdigital package.
    This package controls the digital card of the blue processing
    electronics (BPE).

---------------------
package DETDIGITAL is
---------------------


  function LOAD_CENTROID_TABLE(START                  : BOOLEAN;
                               SRC_AND_SEQUENCE_COUNT : UINT16)
                               return BOOLEAN;


      Starts or stops (START = TRUE/FALSE respectively) the loading of the
      Centroid Lookup Table. The table contents are derived from parameters
      supplied by an earlier call to SET_TABLE_BOUNDARIES.

      Returns TRUE if the command is accepted.

  function SET_TABLE_BOUNDARIES(X_AND_Y_TABLES         : PACKET_CENTROID_TYPE;
                                SRC_AND_SEQUENCE_COUNT : UINT16)
                                return BOOLEAN;


      Specifies the parameters to be used by LOAD_CENTROID_TABLE.

      where:

      X_AND_Y_TABLES(0) = 0 /1 (Disable/Enable) and requests whether the
      table contents should be verified after loading.
      X_AND_Y_TABLES(1->9)   contain the X centroid table boundaries.
      X_AND_T_TABLES(10->18) contain the Y centroid table boundaries.

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

  function LOAD_WINDOW_TABLE(START                  : BOOLEAN;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN;


      Starts or stops (START = TRUE/FALSE respectively) the loading of the
      Window Bitmap Table. The table contents are derived from parameters
      supplied by an earlier call to SET_WINDOW_DESCRIPTION.

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.

  function SET_WINDOW_DESCRIPTION(WINDOW_TABLE           : PACKET_WINDOW_TYPE;
                                  SRC_AND_SEQUENCE_COUNT : UINT16)
                                  return BOOLEAN ;


      Specifies the parameters to be used by LOAD_WINDOW_TABLE.

      where:

      WINDOW_TABLE(0) = 0 /1 (Disable/Enable) and requests whether the
      table contents should be verified after loading.
      WINDOW_TABLE(1) - the number of windows (N) to be loaded (1->15)
      WINDOW_TABLE(2+(n-1)*4) - the Xlow  coordinate (CCD pixels), window n.
      WINDOW_TABLE(3+(n-1)*4) - the Ylow  coordinate (CCD pixels), window n.
      WINDOW_TABLE(4+(n-1)*4) - the Xsize coordinate (CCD pixels), window n.
      WINDOW_TABLE(5+(n-1)*4) - the Ysize coordinate (CCD pixels), window n.
       NOTE: n is in the range 1->N.

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.
```

```
function INTEGRATION(ENABLE                 : UINT16;
                     SRC_AND_SEQUENCE_COUNT : UINT16)
                     return BOOLEAN;


    ENABLE = TRUE/FALSE will enable/disable the blue detector integration
    (i.e. when events are sent to the DPU). The start is synchronised to
    the next end of frame transfer phase of the CCD.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

    Returns a TRUE value of no errors occur during commanding.

function SET_ACQUISITION_MODE(MODE                   : UINT16;
                              SRC_AND_SEQUENCE_COUNT : UINT16)
                              return BOOLEAN;


    Sets the acquisition mode of the detector.

    MODE can one of the following values:

    Value             Meaning
    -----    ----------------------------
      0      Low Resolution, Windowed.
      1      Low Resolution  Full Frame.
      2      High Resolution, Windowed.
      3      High Resolution, Full Frame.
      4      Engineering, x m/n data.
      5      Engineering, y m/n data.
      6      Engineering, event height.
      7      Engineering, event energy.

    NOTE: 4 and 5 are equivalent, 6 and 7 are equivalent.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

    Returns a TRUE value of no errors occur during commanding.

function SET_EVENT_THRESHOLD(THRESHOLD              : UINT16;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN;


    Sets the threshold of events the detector will accept.

    THRESHOLD gives the value of the peak CCD pixel value above which
    events are detected.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

function GET_EVENT_THRESHOLD return UINT16;


    Returns the value for THRESHOLD (for HK purposes) supplied by an earlier
    call to SET_EVENT_THRESHOLD.

function DISABLE_FRAME_TAG(ON_OFF : BOOLEAN;
                           SRC_AND_SEQUENCE_COUNT : UINT16)
                           return BOOLEAN;


    Controls the insertion of frame tag words into the data stream sent to
    the DPU.

    ON_OFF = TRUE/FALSE = Do NOT Insert/ Do Insert respectively
    (note inversion of normal conventions).

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

    Returns TRUE if the command is accepted.

function RESET_CAMERA_HEAD_ELECTRONICS(SRC_AND_SEQUENCE_COUNT : UINT16)
                                       return BOOLEAN;


    This commands resets the camera head electronics.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
```

```
    associated telecommand.

    Returns TRUE if the command is accepted.

function CAMERA_RUNNING(RUNNING : UINT16;
                        SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN ;


    If RUNNING is TRUE,  the camera mode is set to 'Started'
    If RUNNING is FALSE, the camera mode is set to 'Standbye'. In this mode
    it is possible to load the window bitmap RAM.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

    Returns TRUE if the command is accepted.

function STATUS return UINT16;


    Returns the Blue Processing Electronics status word.
    The contents are as follows:

                                    LSB
    -----------------------------------------
    |    |    |    |    |    |    |    |    |
    | IA |   Int Mode    | FE | XX | TE | ME |
    -----------------------------------------

     IA       - Integration Active = 1.
     Int Mode - as per SET_ACQUISITION_MODE.
     FE       - Frame Tag, 1 = No Frame Tags
     XX       - "Don't Care"
     TE       - 0/1 = BPE/ICU can access Centroid Tables
     ME       - 0/1 = Clocks halted, ICU access Window Bitmap/ Camera Started
```

#### *6.4.2.12 detdigital.adb*

```
Extracted from file "detdigital.adb"


 Function
 ========

 This file contains the body for the detdigital package.
 This package controls the digital card of the blue processing
 electronics (BPE).

 This algorithms used here are derived from the document "Software
 Setup of the Blue Detector Electronics", XMM-OM/MSSL/SP/77.

package body DETDIGITAL is


   The following routines are totally internal to detdigital.

   function MIC_OUT(MACS_ACTION : INTEGER;
                  SUBADR : ICB.SUB_ADDRESS_TYPE;
                  DATUM  : UINT16) return BOOLEAN;

   function TABLE_ADDRESS(M : INTEGER; N : UINT16) return UINT16;

   function MAP_ADDRESS(X : UINT16; Y : UINT16) return UINT16;

   function TABLE_DATA(XSUB : INTEGER; YSUB : INTEGER) return UINT16;

   The following tasks are totally internal to detdigital.

   task LOAD_CENTROID_TABLE_TASK;
   task type LOAD_WINDOW_TABLE_TASK_TYPE;

  function MIC_OUT(MACS_ACTION : INTEGER;
                  SUBADR : ICB.SUB_ADDRESS_TYPE;
                  DATUM  : UINT16) return BOOLEAN is


      This routine performs those functions associated with reading
      or writing to the Instrument Control Bus using the MACSbus protocol.

      Delay a bit if this routine is called a lot to allow other tasks to run

     If the requested action is to write data.

       Write the datum to the supplied sub-address.

       If there was a MACSbus error.

          Increment the error count.

     Otherwise, the action is to verify the datum.

       If it is a request to write to the centroid lookup table.

         Read back the datum from the supplied sub-address instead.

         If the value read back is not the same as the supplied datum.

           Increment the verification error count.

       else if it is a request to write to the window bitmap table.

         Read back the datum from the supplied sub-address instead.

         If the value read back is not the same as the supplied datum.

            Increment the verification error count.

       otherwise, we treat the request as a normal write to the supplied
       sub-address.

       Count any macsbus errors that occurred as well.

     Always return OK.

  function TABLE_ADDRESS(M : INTEGER; N : UINT16) return UINT16 is


     Construct a centroid lookup table address from the supplied M,N values
```

```
   Address = N ored with (M shifted left by 8 places)

function MAP_ADDRESS(X : UINT16; Y : UINT16) return UINT16 is


   Construct a window bitmap table address from the supplied X,Y values
   Address = X ored with (Y shifted left by 8 places)

function TABLE_DATA(XSUB : INTEGER; YSUB : INTEGER) return UINT16 is


   Construct a centroid lookup table datum from the supplied x and y
   Sub Pixel values (XSUB and YSUB).
   Datum  = XSUB ored with (YSUB shifted left by 4 places)

task body LOAD_CENTROID_TABLE_TASK is


 This ADA task loads up the centroid lookup table in the BPE.

   Commence infinite loop

     Await a request to start processing.

     Convert stored uplinked boundary values to actual values

     Determine if we are also verifying the data

     Zero error counts

     Now start a maximum of 2 passes (write + optional verify )

        First enable the centroid lookup table for ICU access using MIC_OUT

        Now begin outer loop over all values of M

           Check whether an abort instruction has come in

              If it has

              exit from loop over M

              Otherwise, do nothing

           Load the initial table address to write to using MIC_OUT
           and TABLE_ADDRESS (and rely on auto-inc AFTERWARDS)

           Now commence loop over all useful values of N

              Calculate equivalent fractional position (with blurring)
              for this M,N combination.

              Find the sub-pixel values for the x table

              Find the sub-pixel values for the y table

              Output the resulting sub-pixel data (using MIC_OUT and TABLE_DATA)
              to the current table location (note that the location written to will
              auto icrement by one after this write).

        Finally, disable for ICU access using MIC_OUT

     If there were no errors during the load

       Send Ground an appropriate event packet.

     Otherwise

       Send an appropriate exception packet to ground.

task body LOAD_WINDOW_TABLE_TASK_TYPE is


 This ADA task loads up the window bitmap table in the BPE.

   This code is based on the algorithm described in "software Setup
     of the Blue Detector Electronics", XMM-OM/MSSL/SP/77.

      Await Start request

     Initialise counters and assume default minimum and maximum row pairs.

     Loop over all possible windows
```

```
  if window active

    Increment count of active windows

    Scale, copy and convert uplinked window info to
    high, low pixel pair units for this window.

    Assign Window ID to this window.

    Check whether this is minimum row pair so far.

    Check whether this is the maximum row pair so far.

We have now
1) Determined the number of valid/active windows,
2) Scaled window parameters to pixel pair units
3) Determined the maximum and minimum row pair used

Now proceed to load the window bitmap table.

Zero error counts

If we have no active windows, exit from task

Determine if we are also verifying the data

Now start (maximum 2) passes (write + optional verify )

Perform initialisations prior to table loading

Enable MIC table for loading using MIC_OUT

Begin loop over used row pairs

  First check whether an abort instruction has come in

    If it has

    exit from loop over row pairs.

    Otherwise, do nothing.

  Load up default window of zero (i.e no window) for all of this row pair.

  Set default row action of vertical transfer if this row pair
  does not intersect any windows.

  If current row pair is greater than or equal to the
  minimum row pair used by the windows

    We can now look for windows intersected by this row pair

    Loop over all active windows */

      If this window is intersected by the current row pair

        Loop over the column pairs within crossed window

          If we are at 1st row pair of a window,...

            Do nothing

            Otherwise assign the window ID to this column pair.

          Change the row action code to indicate the presence of a window.

  If the row action code indicates a window intersection.

    Calculate where the row pair is in the block of window intersections.

    otherwise, we have left a block, so reset the calculation.

  now write appropriate action code for current row pair using MIC_OUT
  and MAP_ADDRESS.

  Determine if we need to rewrite PRIOR row pair action code

  If we are at the 1st row pair within a window intersection block,
  and it's not the first row pair overall

    We are about to rewrite a value in a table.
    However, if we are in the compare phase
```

```
          Cancel the previous error caused by an earlier mismatch
          caused by the next instruction.

        Rewrite action code as 'readout and dump' for prior row using MIC_OUT
        and MAP_ADDRESS.

      Now, if it's a a READOUT row, output the window ID's noted earlier using MIC_OUT
      and MAP_ADDRESS.

      Move on to next row pair

   Finally, load final 2 rows of action codes (always 'Vertical Transfer'
   followed bt 'Terminate and Skip') using MIC_OUT and MAP_ADDRESS.

    Make MIC ready for use by starting the camera using MIC_OUT.

    If there are no errors.

     Send ground a suitable event report.

    Otherwise

      Send ground a suitable exception report.

function LOAD_CENTROID_TABLE(START                   : BOOLEAN;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN is


   Attempt to start the load centroid table task
   LOAD_CENTROID_TABLE_TASK

     Return a success condition if it is accepted.

     send the ground an unsuccessful command message.

     and return a failure exit condition.

function SET_TABLE_BOUNDARIES(X_AND_Y_TABLES         : PACKET_CENTROID_TYPE;
                              SRC_AND_SEQUENCE_COUNT : UINT16)
                              return BOOLEAN is


   Save the supplied params

   In this release, simply return TRUE. Future releases should check
   validity of params and return FALSE and issue an invalid command
   acceptance packet

function LOAD_WINDOW_TABLE(START                   : BOOLEAN;
                           SRC_AND_SEQUENCE_COUNT : UINT16)
                           return BOOLEAN is


    Attempt to start the load window bitmap table task
    LOAD_WINDOW_TABLE_TASK

    Return a success condition if it is accepted.

   but if there is no response after a while

     Send the ground an unsuccessful command message.

     and return a failure exit condition.

function SET_WINDOW_DESCRIPTION(WINDOW_TABLE           : PACKET_WINDOW_TYPE;
SRC_AND_SEQUENCE_COUNT : UINT16)
                                return BOOLEAN is


    Save the supplied params

   Set up valid window flags for windows for which data was supplied.

   In this release, simply return TRUE. Future releases should check
   validity of params and return FALSE and issue an invalid command
   acceptance packet

function INTEGRATION(ENABLE                 : UINT16;
                     SRC_AND_SEQUENCE_COUNT : UINT16)
                     return BOOLEAN is
```

```
   Note: We allow integration enabling only if in science or engineering mode
   but always allow disabling

      If the above conditions are true, perform the requested action
      using MIC_OUT.

      Return a success condition (TRUE).

   Otherwise

      Send a suitable command execution failure

     Return a failure condition (FALSE).

function SET_ACQUISITION_MODE(MODE                   : UINT16;
                              SRC_AND_SEQUENCE_COUNT : UINT16)
                              return BOOLEAN is


    Ensure supplied acquisition mode is in range and merge with
    last value of frame tag requested (because they share the same port).

    Send the appropriate command the detector electronics
    using MIC_OUT.

    In this release, return success flag.
    In the event of a MACSbus error, we should send a command
    failure , however the MACSbus error count in HK will increase
    instead.

function SET_EVENT_THRESHOLD(THRESHOLD              : UINT16;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN is


   Load up 2's complement of supplied threshold (as required by Detector
   electronics) via the ICB MACSbus using MIC_OUT.

   If no errors

     Store the threshold value requested.

   In this release, always return success flag.
   In the event of a MACSbus error, we should send a command
   failure message, however the MACSbus error count in HK will increase
   instead

function GET_EVENT_THRESHOLD return UINT16 is


   Return the threshold value store by SET_EVENT_THRESHOLD.

function DISABLE_FRAME_TAG(ON_OFF : BOOLEAN;
                       SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN is


    Merge frame tag setting requested with last value of acquisition
    (because they share the same port).

    Send to the appropriate port via the ICB MACSbus using MIC_OUT.

    In this release always return a success condition.
    (In the event of a MACSbus error, we should send
    an execution failure message, however the MACSbus error count in HK
    will increase instead)

function RESET_CAMERA_HEAD_ELECTRONICS(SRC_AND_SEQUENCE_COUNT : UINT16)
                                       return BOOLEAN is


   Send the appropriate command to the appropriate port via the ICB MACSbus
   using MIC_OUT.

   In this release always return a success condition.
   (In the event of a MACSbus error, we should send
   an execution failure message, however the MACSbus error count in HK will
   increase instead)

function CAMERA_RUNNING(RUNNING : UINT16;
                        SRC_AND_SEQUENCE_COUNT : UINT16)
                                       return BOOLEAN is
```

```
    If the request is to start the camera.

     Send the appropriate command to the appropriate port via the ICB MACSbus
     using MIC_OUT.

   Otherwise

     Send the appropriate command to the appropriate port via the ICB MACSbus
     using MIC_OUT to place it in standby.

    In this release always return a success condition.
    (In the event of a MACSbus error, we should send
    an execution failure message, however the MACSbus error count in HK
    will increase instead)

function STATUS return UINT16 is


    Get the Word containing the status word from the appropriate sub-address
    on the ICB MACSbus.

    Extract and return the relevant bits.
```

#### *6.4.2.13 detector.ads*

Extracted from file "detector.ads"

```
    Function
    ========

    This file contains the specification for the detector package.

    It effectively acts as a 'wrapper' for two other packages,
    DET_DIGITAL controlling an monitoring the digital functions
    of the detector electronics, while DET_ANALOG is the analogue equivalent.
    This is to provide a common interface.

---------------------
package DETECTOR is
---------------------


  function SET_FINE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                            SRC_AND_SEQUENCE_COUNT : UINT16)
                                            return BOOLEAN
    renames DETANALOG.SET_FINE_POSITION_SENSOR_CURRENT;


  function FINE_SENSOR_CURRENT return UINT16
    renames DETANALOG.FINE_SENSOR_CURRENT;


  function SET_FLOOD_LED_BIAS_CURRENT( LED : in UINT16;
                                       SRC_AND_SEQUENCE_COUNT : UINT16)
                                       return  BOOLEAN
    renames DETANALOG.SET_FLOOD_LED_BIAS_CURRENT;


  procedure SET_HV_ENABLE(ENABLED : BOOLEAN)
    renames DETANALOG.SET_HV_ENABLE;


  procedure SET_HV(HV        : HV_TYPE;
                   VALUE     : UINT16)
    renames DETANALOG.SET_HV;


  function LOAD_HV_RAMP_PARAMETERS(VOLTAGE : UINT16;
                                   VALUE : UINT16;
                                   RAMP_RATE : UINT16;
                                   FORCE : UINT16;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN
    renames DETANALOG.LOAD_HV_RAMP_PARAMETERS;


  function HV_RAMP_START(SRC_AND_SEQUENCE_COUNT : UINT16)
                         return BOOLEAN
    renames DETANALOG.HV_RAMP_START;


  function HV_RAMP_STOP(SRC_AND_SEQUENCE_COUNT : UINT16)
                        return BOOLEAN
    renames DETANALOG.HV_RAMP_STOP;


  function PERFORM_HV_SAFING(LEVEL : UINT16;
                                                                               SRC_SEQ_COUNT :
UINT16)
                                                                               return BOOLEAN
    renames DETANALOG.PERFORM_HV_SAFING;


  function SAFE_ONE_HV(VOLTAGE : HV_TYPE;
                       SRC_SEQ_COUNT : UINT16)
                       return BOOLEAN
    renames DETANALOG.SAFE_ONE_HV;


  function GET_SET_GO(VOLTAGE : HV_TYPE) return INTEGER
    renames DETANALOG.GET_SET_GO;


  procedure SET_ADC_ACCURACY(ACCURACY : UINT16)
```

```
    renames DETANALOG.SET_ADC_ACCURACY;


  function  GET_ADC_ACCURACY
    return UINT16
    renames DETANALOG.GET_ADC_ACCURACY;


  function GET_ANALOG(ADC_ITEM : UINT16)
                    return UINT16
    renames DETANALOG.GET;


  function FINE_POSITION_SENSED
    return BOOLEAN
    renames DETANALOG.FINE_POSITION_SENSED;


  function FLOOD_LED_BIAS_CURRENT
    return UNIBBLE
    renames  DETANALOG.FLOOD_LED_BIAS_CURRENT;


  function HV_ENABLED
    return BOOLEAN
    renames DETANALOG.HV_ENABLED;


  function LOAD_CENTROID_TABLE(START                  : BOOLEAN;
                               SRC_AND_SEQUENCE_COUNT : UINT16)
                               return BOOLEAN
    renames DETDIGITAL.LOAD_CENTROID_TABLE;


  function SET_TABLE_BOUNDARIES(X_AND_Y_TABLES         : PACKET_CENTROID_TYPE;
                                SRC_AND_SEQUENCE_COUNT : UINT16)
                                return BOOLEAN
    renames DETDIGITAL.SET_TABLE_BOUNDARIES;


  function LOAD_WINDOW_TABLE(START                  : BOOLEAN;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN
    renames DETDIGITAL.LOAD_WINDOW_TABLE;


  function SET_WINDOW_DESCRIPTION(WINDOW_TABLE           : PACKET_WINDOW_TYPE;

          SRC_AND_SEQUENCE_COUNT : UINT16)
                                  return BOOLEAN
    renames DETDIGITAL.SET_WINDOW_DESCRIPTION;


  function INTEGRATION(ENABLE                 : UINT16;
                       SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN
    renames DETDIGITAL.INTEGRATION;





  function GET_EVENT_THRESHOLD return UINT16
    renames DETDIGITAL.GET_EVENT_THRESHOLD;


  function DISABLE_FRAME_TAG(ON_OFF : BOOLEAN ;

SRC_AND_SEQUENCE_COUNT : UINT16)
                                                                              return BOOLEAN
    renames DETDIGITAL.DISABLE_FRAME_TAG;


  function RESET_CAMERA_HEAD_ELECTRONICS(SRC_AND_SEQUENCE_COUNT : UINT16)
                                         return BOOLEAN
    renames DETDIGITAL.RESET_CAMERA_HEAD_ELECTRONICS;


  function CAMERA_RUNNING(RUNNING : UINT16;
                          SRC_AND_SEQUENCE_COUNT : UINT16)
                                                                           return BOOLEAN
    renames DETDIGITAL.CAMERA_RUNNING;
```

```
function DIGITAL_STATUS
  return UINT16
  renames DETDIGITAL.STATUS;

procedure FINE_SENSOR(ON_OFF : BOOLEAN)
  renames DETANALOG.FINE_SENSOR;
```

#### *6.4.2.14 dpu.ads*

Extracted from file "dpu.ads"

```
   Function
   ========

   This file contains the specifications for the DPU package. That package
   controls and monitors the DPU via commands and data records described in
   the ICU-DPU Protocol Document (XMM-OM/MSSL/ML/0011).

--------------
package DPU is
--------------


  function COMMAND(WORD : UINT16_ARRAY;
                   SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN;


   where the array WORD contains the DPU command to be sent to the DPU
   via the SSI interface.

  function HEARTBEATS return UINT16;


   returns DPU heartbeat count since startup. It 'wrapsaround' at 65535.

  function STATUS return UINT16;


   returns the DPU Status word contained in the DPU heartbeat.
   The contents of the status word are defined in the ICU-DPU
   Protocol Document (XMM-OM/MSSL/ML/0011) in the section describing thr
   DA_HBEAT record.

  function DRIFT_X return LONG_INTEGER;


   Returns the drift in x extracted from the most recent DA_TRK record.
   The units are 1/1000 centroided pixels.

  function DRIFT_Y return LONG_INTEGER;


   Returns the drift in y extracted from the most recent DA_TRK record.
   The units are 1/1000 centroided pixels.

  function ROLL return LONG_INTEGER;


   Returns the drift in roll extracted from the most recent DA_TRK record.
   The units are 1000000*sin(roll).

  function FRAME_COUNT return UINT16;


     Returns the frame count for this exposure extracted from the most
     recent DA_TRK record.

  function FRAMES_PER_EXPOSURE return UINT16;


     Returns frames so far for this exposure extracted from
     the most recent DA_BEGOF_EXP record.

  function EXPOSURE_ID return LONG_INTEGER;


     Returns the Exposure ID contained in the most recent DPU heartbeat.

  function DATA_ALERTED return UINT16;


     Returns the ID of the type of science data (DD_xxx records) that
     is currently being processed. This information is extracted from
     the DA_DATA_ALERT record.

  task HEARTBEAT_WATCHDOG is

    entry START;
```

```
   entry STOP;
   entry RESET;

end HEARTBEAT_WATCHDOG;


    This task monitors the DPU heartbeat and issues an appropriate Exception
    if it stops.

    HEARTBEAT_WATCHDOG.START starts the heartbeat monitoring task.
    HEARTBEAT_WATCHDOG.STOP stops the heartbeat monitoring task.
    HEARTBEAT_WATCHDOG.RESET effectively stops then starts the heartbeat
    monitoring task in order to reset its internal timout timers.

procedure INIT;


    Initialises the SSI hardware interface and starts the data monitoring
    task.

procedure BENT_PIPE(ENABLE : BOOLEAN);


    Enable/disables the 'bent-pipe' diagnostic - this ensures that all
    DPU data records are sent out as packets, even when the corresponding
    packets types are disabled.

procedure ENABLE_REQ_DATA(ACTION : BOOLEAN);


    Enable/disables (ACTION = TRUE = Enabled) the icu-dpu 'handshake'
    which automatically ensures that DD_xxx blocks and DR_xxx blocks are
    send on to ground as soon as they are available.

procedure SET_FILTER(MODE : UINT16);


    Inform DPU of current filter MODE in use.

procedure POWER_DOWN;


    Power Down the DPU.

procedure SYNCH_CLOCK(SECS : UINT16);


    Inform DPU of spacecraft time to the nearest second (contained in
    SECS) on the occurence of next BCP2/4 pulse.

procedure ABORT_EXP;


    Abort current exposure

procedure INIT_DPU;


    Init DPU (zeroes memory, readies swap units - a "Dave"

procedure DISABLE_SSI_OUTPUT(DISABLED : BOOLEAN);


    Disable all SSI output except Heartbeats
```

#### *6.4.2.15 dpu.adb*

```
Extracted from file "dpu.adb"


 Function
 ========

 This file contains the body for the DPU package. The package
 controls and monitors the DPU via commands and data records described in
 the ICU-DPU Protocol Document (XMM-OM/MSSL/ML/0011). All data structures
 used in this package are implicitly defined inthat document.

-------------------
package body DPU is
-------------------


   Create buffer to hold all data received from DPU

   Create buffer to hold DP_WDW derived info

   Create buffer to hold DD_ENG derived info

   Define routines/tasks specifications internal to the package.

  task DATA_MANAGER is
    pragma priority(IMPORTANCE.DPU_DATA_MANAGER);
    entry START;
  end DATA_MANAGER;


   where the DATA_MANAGER task monitors ALL data from the DPU
   and takes appropriate action (e.g.  counts heartbeats etc).

  procedure REQ_DATA;


   where REQ_DATA causes a request for 1 block of data to
   be sent to the DPU. This is only meaningful after receiving a
   DA_DATA_ALERT from the DPU

   Define the bodies of internal routines/tasks

  task body HEARTBEAT_WATCHDOG is


     Start infinite loop

        Await a call to an entry point.

          If a call to the RESET entry is made,
          this resets the timout count.

        Or

           If call to the START is made, start the
           DPU heartbeat watchdog monitor.

         Or

           If call to STOP is made, stop the DPU heartbeat watchdog.

         Otherwise

           Provided the task is set to be running

             and nothing is done for timout period (30 sec)

             send a DPU Heartbeat Exception packet.

  procedure REQ_DATA is


     causes a request for 1 block of data to
     be sent to the DPU. This is only done after receiving a
     DA_DATA_ALERT from the DPU

     If the ICU-DPU science data 'handshake' is enabled (the default)

       Send an IC_REQ_DATA command to the DPU via the SSI interface.
```

```
      Wait a bit

    Otherwise

      Set the data pending flag.

procedure ENABLE_REQ_DATA(ACTION : BOOLEAN) is


    If we are enabling the science data handshake and data is pending

      Request it using aan IC_REQ_DATA DPU command.

      and clear the data pending flag

    Store requested state (enable/disable) of handshake
    for later comparison.

task body DATA_MANAGER is


    This task monitors ALL data from the DPU
    and takes appropriate action (e.g.  counts heartbeats etc).

    In order to follow the logic of this code, you must be aware that
    the data block received from the DPU via the SSI interface has the
    following format

    ++++++++++++++++++++++++++++++++++++++++++
    + Word 0 + Word 1 + Word 2  ->  Word N+2 +
    ++++++++++++++++++++++++++++++++++++++++++
    + Block  + Word   +                      +
    +  Type  + Count  +    DPU Data Block    +
    +        +   N    +                      +
    ++++++++++++++++++++++++++++++++++++++++++

    Wait for start instruction from main program to synchronize with
    other code.

    Start DPU Heartbeat Watchdog using HEARTBEAT_WATCHDOG.START.

    Begin infinite loop

       Begin second infinite loop

         Get the next DPU block using SSI_IN.GET

         only exit from loop if it's a valid block.

       Extract the block type from the 1st word

          If it's priority science data block (i.e. DP_xxx block type)

             If appropriate SID for this block type is enabled

               Forward to the priority data output routine in
               the SCIENCE_FM package.

             If it's a DP_WDW record

                and the ICU is not in engineering mode

                  Provided we have between 1 and 15 windows
                  (some DPU eng modes have > 15)

                    Then we need to set up the detector electronics
                    from information stored in the DP_WDW record.

                    1) loop over the windows decribed in the record,
                    extracting the x0, y0, xsize, ysize
                    parameters for the detector windows contained in the
                    DP_WDW record
                    2) scale them to CCD pixels (which is a
                    function of the BPE binning to be used in the exposure
                    and was extracted earlier from a IC_BPE_BINNING command)
                    3) add the active area offset (which is a function of
                    whether this is the prime or redundant half).

                    4) Load up the Window Bitmap tables in MIC
                    to correspond to these detector windows using
                    the DETECTOR package.

          If it's regular science data (i.e. a DD_xxx block type)
```

```
    Determine SID associated with this particular DD_xxx block
    from a lookup table.

    Forward Regular data if appropriate SID is enabled

      Forward to the regular data output routine in
      the SCIENCE_FM package.

    If it's a DD_ENG record

      Count how many DD_ENG records so far.

      If it's the 1st DD_ENG record after a DA_ALERT saying
      DD_xxx data is available, then check if it's the
      channel boundary data (sub-type 3).

        Set flag forcing data will be verified

        Extract the channel boundaries from the DP_XXX record

        Load up the MIC centroid tables accordingly using the
        DETECTOR package.

  If it's an alert (i.e. a DA_xxx record)

    Determine default NHK sub-type (event or exception)
    from command code
    and SID associated with this DA_xxx block from look-up table.

    (Now perform actions that are alert specific)

    If it's a heartbeat

      Count heartbeats (wrapping around if necessary)

      Reset heartbeat watchdog to prevent a timout
      using HEARTBEAT_WATCHDOG.RESET.

      Extract DPU Status Word from heartbeat and store.

      Correct for DPU ROM bug (as per NCR 89)

      Determine from status word which DPU code we are running.
      (i.e. 'Fred' (ROM code) or 'Jim' (Uplinked Code))

      Extract Exposure ID from the heartbeat record.

      Inform waiting filter wheel movement request
      (if any) that h/beat has occurred
      using MECHANISM. AWAIT_DPU_HEARTBEAT.

    If it's a 'Fred' (DA_DPU_BOOT_READY)
    i.e. we have just started running the
    the DPU ROM code.

      If we were not expecting one (i.e. no preceding IC_RESET_DSP)

        Note that the NHK packet will be a major anomaly,
        and change the SID accordingly

      Ensure any prior mem dumps that might have been in
      progress are flushed (NCR 182) using MEMDPU.FLUSH.

      Similarly, ensure any science data group currently
      being dumped is flushed (NCR 182) using SCIENCE_FM.FLUSH.

    If it's a 'Jim' (DA_DPUOS_READY) - i.e. we have just started
    running the uplinked DPU code.

        Ensure engineering record (DD_ENG) data counters are reset.

    If it's a clock sync error (DA_CLK_SYNCH_ERROR) block

      Extract the commanded and previous times from DPU block

      If the commanded time is the same as the old time
      we will note that its associated NHK packet event will be
      an event rather than an exception (and modify SID accordingly)

    If it's a data alert (DA_DATA_ALERT)

      Note which type of regular data we have an alert for
```

```
          for use when we process the DA_DATA_END block later.

          Request 1 block of data via the REQ_DATA routine

        If it's a data_end (DA_DATA_END)

          If it's the end of regular science data
          (deduced when we processed the DA_DATA_ALERT)

            Flush the current science packet group buffer
            (via SCIENCE_FM.FLUSH)

            and also reset the DD_ENG record counter as failsafe.

          If it's the end of RAM/ROM dump packets (DR_xxx blocks)

              Flush out the current memory dump packet buffer
              (via MEMDPU.FLUSH).

          Otherwise, do nothing

          Clear the datatype flag which notes which type of regular
          data is being processed.

        If it's a DPU_MNEMO.DA_TRK alert

          Extract the current frame count from the record.

          Extract the drift information from the record.

        If it's a DA_BEGOF_EXP

          Extract the frames for this exposure from the record.

        If it's a ENDOF_EXP

          Ensure detector integration is turned off
           using DETECTOR.INTEGRATION.

        If it's a multi-bit error

          Reset level of associated NHK report to Major Anomaly

        (Now do things that are generic to all alerts)

        Forward all alerts as auxiliary data packets if enabled
        via SCIENCE_FM.AUXILIARY_DATA

        Possibly send to ground as an NHK packet (event or exception)
        via NHK.PUT but only if SID is enabled
        (whether a given SID is enabled is decided internally by
        the package NHK, and thus whether the packet is actually sent)

      If they are memory dump blocks (DR_xxx)

          Output them (via MEMDPU.PUT) as memory
          Dump packets

      If they are anything else

        Do nothing.

 Define bodies of externally visible tasks/procedures

function COMMAND(WORD : UINT16_ARRAY;
                  SRC_AND_SEQUENCE_COUNT : UINT16
                ) return BOOLEAN  is


   Reserve memory for command buffer.

   Loop over the number of words in the command
   (derived from the second location of the input command)..

     and copy the command words into a temporary command buffer

   If it's a zero length IC_SYNCH_CLK time sync command.

     Wait for next BCP4 pulse, and get On-Board-Time
     (via Time Manager package)

     Extract Secs field from the On-Board-Time
```

```
      Add one to it, with possible wraparound, to deduce time at next BCP4.

      Modify the IC_SYNCH_CLK command in the temporary command buffer
      accordingly by restoring it to its correct length and adding in
      the least 14 sig bits of the seconds field derived above.

   If it's a 'Fred' (IC_RESET_DSP) command.

       Reset DPU heartbeat watchdog using HEARTBEAT_WATCHDOG.RESET.

      Set a flag indicatiing we now expect a 'Fred' (DA_DPU_BOOT_READY)

   If it's a 'Jim' (IC_LOAD_DPUOS) command.

       Reset heartbeat watchdog using HEARTBEAT_WATCHDOG.RESET.

   If it's an engineering mode command (IC_ENBL_ENG)

      Check whether the ICU is not in engineering mode

         and reject with a "Invalid for this Mode" message.

         Return with a failure condition of FALSE.

   If it's an Set BPE Binning command (IC_BPE_BINNING)

      Extract the requested BPE binning for later use
      when processing the DP_WDW record..

   Send temporary command buffer to the DPU via SSI.PUT.

   Return a success condition.

function HEARTBEATS return UINT16 is


   Return the heartbeat count deduced when processing the heartbeats.

function DRIFT_X return LONG_INTEGER is


   Return the drift in X extracted from DA_TRK.

function DRIFT_Y return LONG_INTEGER is


   Return the drift in Y extracted from DA_TRK.

function ROLL return LONG_INTEGER is


   Return the drift in Roll extracted from DA_TRK.

function FRAME_COUNT return UINT16 is


   Return the frame so far this exposure extracted from DA_TRK.

function FRAMES_PER_EXPOSURE return UINT16 is


   Return the Total Frames for this exposure extracted from DA_BEGOF_EXP.

function EXPOSURE_ID return LONG_INTEGER is


   Return the Exposure ID extracted from the heartbeat record.

function STATUS return UINT16 is


   Return the DPU Status Word extracted from the heartbeat record.

function DATA_ALERTED return UINT16 is


   If the Block ID of the regular data currently being 'handshaked'
   corresponds to regular science (DD_xxx)

     Return that ID

   Otherwise
```

```
     Return 0

procedure INIT is

    Initialize the SSI Card and Controlling Software
    using SSI_OUT.RESET.

    Start the DPU Data Manager processing the DPU output
    using DATA_MANAGER.START.

procedure SET_FILTER(MODE : UINT16) is

   Construct an IC_LOAD_FILT_CONF with filter set according to
   the value MODE.

    Provided the DPU is not in boot mode

       Send the command to the DPU.

procedure POWER_DOWN is

    Construct an IC_POWER_DOWN_DOWN command.

   Send the command to the DPU via the SSI interface
   using SSI_OUT.PUT.

procedure SYNCH_CLOCK(SECS : UINT16) is

   Construct an IC_SYNCH_CLK using the value SECS accordingly.

   Send it to the DPU using SSI_OUT.PUT.

procedure ABORT_EXP is

    Construct an IC_ABORT_DPU command.

procedure INIT_DPU is

    Construct an IC_INIT_DPU command.

procedure DISABLE_SSI_OUTPUT(DISABLED : BOOLEAN) is

    Construct an IC_LOCAL_RAM  command.
```

#### *6.4.2.16 dpu_mem_manager.ads*

Extracted from file "dpu_mem_manager.ads"

```
   function LOAD_MEMORY(MID: UINT16;
                        START_ADDRESS: LONG_INTEGER;
                        DATA: UINT16_ARRAY;
                        LENGTH: UINT16;
                        SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN;


       where MID is the MID
       where START_ADDRESS is the start address of the load
       where DATA is the data to load as an array of unsigned 16 bit words
       where LENGTH is the length of the data in words
       where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
       returns a boolean: true on success and false on failure
       function LOAD_MEMORY loads memory corresponding to the MID

   function DUMP_MEMORY(MID: UINT16;
                        ADDRESS: LONG_INTEGER;
                        LENGTH: UINT16;
                        SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN;


       where MID is the MID
       where ADDRESS is the address of the dump request
       where LENGTH is the length of the requested memory dump in words
       where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
       returns a boolean: true on success and false on failure
       function DUMP_MEMORY dumps memory corresponding to the MID
```

#### *6.4.2.17 dpu_mem_manager.adb*

```
Extracted from file "dpu_mem_manager.adb"


 Dependencies
 ============

with INTRINSICS;
with UNCHECKED_CONVERSION;
with ARTCLIENT;
with SYSTEM;

with PACKET;
with TC_VERIFY;
with TMQ;
with PEEK_POKE;
with CRC;
with DPU_MNEMO;
with SSI_OUT;
with DEBUG;
with NHK;
with MEMLOC;


-------------------------------
package body DPU_MEM_MANAGER is
   -------------------------------


   function DUMP_MEMORY(MID: UINT16; ADDRESS: LONG_INTEGER; LENGTH: UINT16;
SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


       returns array 0 .. packet.MAX_TM_MEM_PARAMS_M1

             DPU local RAM
             If length is out of range, send an error packet

             If address is out of range, send an error packet

                If the address is OK, form an SSI block

                and send block down SSI

             DPU global memory 24-bit words
             If address is out of range, send an error packet

                If the address is OK, form an SSI block

                and send block down SSI

             DPU global memory 16-bit words
             If address is out of range, send an error packet

                If the address is OK, form an SSI block

                and send block down SSI

             When the MID is 20-27
             (length is a 16-bit number of 24-bit words to dump)
             If address is out of range, send an error packet

                If the address is OK, form an SSI block

                and send block down SSI

             For other MIDs send unsuccessful acceptance

   function DPU_CHECKSUM(DPU_ARRAY: UINT16_ARRAY) return UINT16 is


       where DPU_ARRAY is an array of words to load into the DPU
       returns the checksum as an unsigned 16-bit integer

       Start with checksum of 0

       For each word starting with the third to the end ...

          Add each byte of the current word to the checksum

       At the end of the block, xor with 0xffff
```

```
      Return the checksum

   function LOAD_MEMORY(MID: UINT16; START_ADDRESS: LONG_INTEGER; DATA: UINT16_ARRAY; LENGTH:
UINT16; SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is

      LENGTH is in 16-bit words

         If the MID is 16#13# load local memory

            For each DPU-word of data

               Form the DPU block

               Remember to convert because DPU uses 3-byte words and we're loading with 2-byte
words

               Put the DPU block down the SSI

            If the MID is 16#14# load global memory (24-bit words)

            For each DPU-word of data

               Form the DPU block

               Remember to convert because DPU uses 3-byte words and we're loading with 2-byte
words

               Put the DPU block down the SSI

            If the MID is 16#15# load global memory (16-bit words)

            For each DPU-word of data

               Form the DPU block

               Put the DPU block down the SSI

            If the MID is 20-27 load program RAM

            Select EEPROM

            Unlock

            For each word of data

               Form the DPU block

                  Remember to convert because DPU uses 3-byte words and we're loading with 2-byte
words

                  Put the DPU block down the SSI

            Lock

            When the MID is wrong

            send unsuccessful acceptance (illegal mid) packet
```

#### *6.4.2.18 dpu_mnemo.ads*

Extracted from file "dpu_mnemo.ads"

```
--------------------
package DPU_MNEMO is
--------------------


    This specification only package contains the values of Command and Data
    mnemonics as defined in the ICU-DPU Protocol Definitions
    XMM-OM/MSSL/ML/0011
```

#### *6.4.2.19 heater.ads*

Extracted from file "heater.ads"

```
    Function
    ========

    This file contains the specification for the HEATER package.
    The algorithms implemented therein are outlined in document
    XMM-OM/MSSL/SP/165. "OM Heater Control"

-----------------
package HEATER is
-----------------


  function SET_MARK_SPACE(HEATER_NO : UINT16;
                          ON_TIME    : UINT16;
                          TOTAL_TIME  : UINT16;
                          SRC_AND_SEQUENCE_COUNT : UINT16
                          ) return BOOLEAN;


      where

      HEATER   specifies heater to be controlled by the open loop algorithm

      ON_TIME  specifies the number of 10 seconds the heater should be on

      OFF_TIME specifies the number of 10 seconds the heater should be off

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.

      NOTE : This function has been superceded by SET_FUNCTION and is no longer
             used.

  function SET_FUNCTION(FID : UBYTE;
                        PARAM1   : UINT16;
                        PARAM2   : UINT16;
                        PARAM3   : UINT16;
                        SRC_AND_SEQUENCE_COUNT : UINT16
                       ) return BOOLEAN;


      This function specified how each heater is to be controlled by which
      automatic algorithm as follows:

      FID  Heater    Description      PARAM1      PARAM2      PARAM3
      ---  ------    -----------      ------      ------      ------
       1  Interface  Closed, Free      Tmin        Tmax         -
       2  Interface  Open, Free       On Time     Cycle Time    -
       3  Forward    Closed, Synched   Tmin        Tmax       Thermistor
       4  Forward    Closed, Free      Tmin        Tmax       Thermistor
       5  Forward    Open, Synched    On Time     Cycle Time    -
       6  Forward    Open, Free       On Time     Cycle Time    -
       7  Focussing  -/+ Focussing    On Time     Cycle Time  Direction
       8    -        Set Sample Time  Sample Time     -         -

      Notes:
       1) On Time and Cycle Time are in units of Sample Time.
       2) Thermistor = 0/1 = Prime/Redundant forward thermistor.
       3) Tmin and Tmax are in 'raw' units.
       4) Focus Direction = -ve = HTR4(Secondary)
                          =  0  = HTR3 and HTR4 off.
                          = +VE = HTR3(Metering) powered.

       5) Sample Time is in units of seconds.

      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

      Returns TRUE if the command is accepted.

  function START return BOOLEAN;


      Starts the automatic heater control algorithms.
```

```
function STOP return BOOLEAN;


    Stops the automatic heater control algorithms

function LOAD_CONFIG_DIRECTLY(CONFIG                 : UINT16;
                              SRC_AND_SEQUENCE_COUNT : UINT16 )
                              return BOOLEAN;


    The bit pattern in CONFIG specifies which heater should be on or off
    ( 1 = on ) as follows:
                                              L.S.B.
    -------------------------------------------------
    | Temperature Control  |        Focussing       |
    -------------------------------------------------
    |    Main    |  Forward   | Metering  | Secondary |
    |            |            |   Rods    |  Mirror   |
    |   (HTR 1)  |  (HTR 2)   |  (HTR 3)  |  (HTR 4)  |
    -------------------------------------------------

    NOTE: This command is ignored if the automatic heater algorithms
          are running.

    SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
    associated telecommand.

    Return TRUE in this release

procedure BRIEF_DISABLE(ENABLE : BOOLEAN);


    If ENABLE = TRUE, turns off all heaters.

    If ENABLE = FALSE, restores prior configation of heaters if the
    automatic algorithms are NOT running, otherwise resumes the automatic
    algorithms.

function CONFIG return UINT16
  renames TMPSU.HEATER_CONFIG;


    Renames, for convenience, the TMPSU package function that returns
    the current heater configuration.

    The bit pattern in CONFIG specifies which heater is on or off
    ( 1 = on ) as follows:
                                              L.S.B.
    -------------------------------------------------
    | Temperature Control  |        Focussing       |
    -------------------------------------------------
    |    Main    |  Forward   | Metering  | Secondary |
    |            |            |   Rods    |  Mirror   |
    |   (HTR 1)  |  (HTR 2)   |  (HTR 3)  |  (HTR 4)  |
    -------------------------------------------------
```

#### *6.4.2.20 heater.adb*

Extracted from file "heater.adb"

```
 Function
 ========

 This file contains the body for the HEATER package.
 The algorithms implemented therein are outlined in document
 XMM-OM/MSSL/SP/165. "OM Heater Control"

   Define Specification for Tasks and Procedures used internally.

  task CONTROL is

    pragma priority(IMPORTANCE.THERMAL_CONTROL);

    entry START;
    entry STOP;
    entry SET_ON_OFF(HEATER_NO : UINT16;
                     ON_TIME   : UINT16;
                     TOTAL_TIME  : UINT16);

  end CONTROL;


   START starts automatic heater control (open or closed loop)
   STOP stops automatic heater control (open or closed loop)
   SET_ON_OFF specifies on/off time when in open loop control

   Note: Default heater/algorithm settings are:

   I/F Heater limits are 19.5 +/- 0.5 under closed loop control
   Forward Heater limits are 19.5 +/- 1.5 under closed loop control
   Focussing heaters are off under open loop control.

  procedure CHANGE_CONFIG(NEW_CONFIG : UINT16);


   Changes the heater configuration to 4 lsb of NEW_CONFIG
   (1 = ON).

   Now specify bodies for internal routines and tasks.

  task body CONTROL is


     Begin infinite loop

         If a call to the START entry point is made

           Get current time.

           Start task running.

           Reset the 'cycle' counter.

           Obtain last known heater configuration using TMPSU.HEATER_CONFIG.

         Or if a call to the STOP entry point is made

           Ensure all heaters off using CHANGE_CONFIG.

           Remember that configuration.

           Then stop algorithm

         Or if a call to the heater parameter entry point is made

           store length of ON time for specified heater.

           store length of duty cycle for specified heater.

         Or, provided heating algorithm is already running

           delay until start of next 'Sample Time'.

           Commence loop over the heaters

             If the open loop algorithm is active for this particular heater
```

```
            Provided we have a non-zero 'cycle time'

              Determine where we are in the cycle for this heater.

              Set flag indicating whether the heater should be on or off.

            otherwise

              Set flag indicating that the heater is off.

          Otherwise we have a closed loop algorithm

            Determines whether the heater is already on from the last
            known configuration.

            If we are dealing with the forward heater.

              Get the control temperature from the specified thermistor

            Otherwise

              Get the control temperature from an average of
              MAIN, REF A and REF B thernistors.

            If heater was on

              and control temperature is above maximum allowed.

                Set flag indicating that the heater should be turned off.

            Otherwise

              If the control temperature  is below minimum allowed.

                Set flag indicating that the heater should be turned on.

        If synchronisation of heaters is enabled

          Enable forward switch on if interface heater is flagged as being
          about to be switched off

          If a switch on (from off) of the forward heater has been
          requested by the automatic algorithm.

            Only flag as allowed if forward switch on is enabled

        Determine resulting heater configuration from flags set.

        Request the TMPSU to command the heaters accordingly
        using CHANGE_CONFIG.

        Remember this configuration for comparison next time.

        Calculate time of next sampling of thermistors

        Count no of heater cycles

 Now specify bodies for external routines and tasks.

function SET_MARK_SPACE(HEATER_NO : UINT16;
                        ON_TIME   : UINT16;
                        TOTAL_TIME  : UINT16;
                        SRC_AND_SEQUENCE_COUNT : UINT16
                        ) return BOOLEAN is


   Specify On time within Cycle Time for specified heater.
   NOTE: This function now obsolete and no longer called.

function SET_FUNCTION(FID : UBYTE;
                      PARAM1   : UINT16;
                      PARAM2   : UINT16;
                      PARAM3   : UINT16;
                      SRC_AND_SEQUENCE_COUNT : UINT16
                     ) return BOOLEAN is


   If the function specified is "Interface, Closed Loop, Free Running"

     then store that fact together with the temperature limits.

   If the function specified is "Interface, Open Loop, Free Running"
```

```
      Then store that fact together with the on and total times.

    If the function specified is "Forward, Closed Loop, Synched"

      Then store that fact together with the temperature limits and thermistor
      to be used.

    If the function specified is "Forward, Closed Loop, Free Running"

      Then store that fact together with the temperature limits and thermistor
      to be used.

    If the function specified is "Forward, Open Loop, Synched"

      Then store that fact together with the on and total times.

    If the function specified is "Forward, Open Loop, Free Running"

      Then store that fact together with the on and total times.

    If the function specified is "Focussing"

      If the focus direction is zero

        Then ensure both focussing heaters will be off.

      If the focus direction is greater than zero

        Then store that the metering rods heater will be on for
        the specified times.

      If the focus direction is less than zero

        Then store that the secondary mirror heater will be on for
        the specified times.

    Otherwise, if we a resetting the sample time.

      Store the new value.

    And for any other values of FID

      Return a failure condition of FALSE.

    The above stored values will be acted upon
    at the start of the next 'Sample Time'

    Return a Success condition of TRUE.

function START return BOOLEAN is


    Start the automatic heater control algorithms
    using CONTROL.START.

    Return Success condition.

function STOP return BOOLEAN is


    Stop the automatic heater control algorithms
    using CONTROL.STOP.

    Return Success condition.

function LOAD_CONFIG_DIRECTLY(CONFIG                 : UINT16;
                              SRC_AND_SEQUENCE_COUNT : UINT16 )
                              return BOOLEAN is


    Provided the automatic heater algorithms are not running

      Load the supplied heater configuration via the TMPSU using
      TMPSU.SET_HEATER_CONFIG.

    Return a Success condition.

procedure BRIEF_DISABLE(ENABLE : BOOLEAN) is


    Provided we did not perform the requested action last time

      If we wish to pause the heater algorithm(s)
```

```
        Note whether automatic version is running
        If the automatic version is running
          then stop it using CONTROL.STOP.
        but if we are relying on ground control
          Remember the current config using TMPSU.HEATER_CONFIG.
          Then turn all heaters off
      If we wish to unpause the heater algorithms
        and the automatic version was running
          Restart it using CONTROL.START.
        But if we were relying on ground control
          restore old config using CHANGE_CONFIG.
      Finally, remember what action was requested ready for next call.
procedure CHANGE_CONFIG(NEW_CONFIG : UINT16 ) is

   Remember current config to compare against
   Initialise working config to that of current
   Loop over all heaters
     If this heater has changed in requested configuration
       Wait a bit to avoid switching two heaters together
       Change record of working configuration to new value for this heater
       Now request (via TMPSU.SET_HEATER_CONFIG) the real heater configuration
       become that of the working configuration, thus updating the actual
       configuration for just this heater.
```

#### *6.4.2.21 hk.ads*

Extracted from file "hk.ads"

```
    Function
    ========

    This file defines the specification for the HK package. The package
    acquires and sends the Housekeeping Packets (HK), the contents of
    which are defined in the XMM-OM Telecommand and
    Telemetry Specification document, XMM-OM/MSSL/ML/0010

-------------
package HK is
-------------


  procedure ON;


    This procedure enables the acquisition of the HK packet type

  procedure OFF(HK_WAS_RUNNING : out BOOLEAN);


    This procedure disables the acquisition of the HK.

  procedure  BLOCK(ACTION : BOOLEAN);


    if Action = TRUE, Blocks the HK if active
    if Action = FALSE, restore HK condition to the last call with ACTION set to TRUE
```

#### *6.4.2.22 hk.adb*

```
Extracted from file "hk.adb"


 Function
 ========

 This file defines the body for the HK package. The package
 acquires and sends the Housekeeping Packets (HK), the contents of
 which are defined in the XMM-OM Telecommand and
 Telemetry Specification document, XMM-OM/MSSL/ML/0010

------------------
package body HK is
------------------


  task PROCESS is
    pragma priority(IMPORTANCE.HK_PROCESS);
    entry ON;
    entry OFF(HK_WAS_RUNNING : out BOOLEAN);
  end PROCESS;


   The above is the specification for the internal task that performs the HK
   acquisition

   Entry ON starts the task.
   Entry OFF stops the task
     and returns whether or not it was already stopped.

   Default to current SID is that associated with 10 second interval between
   packets.

  task body PROCESS is


     Default that the task is running.

     Default requested next HK packet to be acquired at current time.

     Create an instance of an HK packet

     Set up initial time delay interval by subtracting current time from next
     requested HK acquisition time.

     Commence infinite loop

       Await for either:

         1) A request to start HK acquisition (already on by default)

           If ON request comes in

           then enable HK acquisition

           Initiliase the next time for HK acquisition to be now

         2) A request to stop HK acquisition

           If OFF request comes in

           then disable acquisition

         3) otherwise, provided HK is enabled (the default)
            and no ON or OFF requests pending

           Wait for the calculated time delay before
           starting to acquire the next HK packet

           Provided the wait interval was not too negative and HK is not blocked

             Ensure HK packet contents zeroed

             If TMPSU secondaries enabled

               Get detector ADC accuracy from DETECTOR.GET_ADC_ACCURACY
               and store in packet.

               Get Thermistor readings from DETECTOR.GET_ANALOG
               and store in packet.
```

```
      Get HV enabled status from DETECTOR.HV_ENABLED and store in packet.

      Get Fine Pos Sensor Status from MECHANISM.AT_FINE_SENSOR
      and DETECTOR.FINE_SENSOR_CURRENT
      and store in packet.

      Get HV values from DETECTOR.GET_ANALOG
      and store in packet.

      Get Low Voltage values from DETECTOR.GET_ANALOG
      and store in packet.

      Get fine pos current from DETECTOR.GET_ANALOG and store in packet.

      Get Flood LED Reading from DETECTOR.FLOOD_LED_BIAS_CURRENT
      and store in packet.

      Get Detector Electronics Status Word from DETECTOR.DIGITAL_STATUS
      and store in packet.

    Get heater status from HEATER.CONFIG and store in packet.

    Get coarse sensor current info from TMPSU.COARSE_SENSOR_CURRENT
    and MECHANISM.AT_COARSE_SENSOR and store in packet.

    Get secondary Voltage status from TMPSU.SECONDARY_VOLTAGES_ENABLED
    and store in packet.

    Get f/w phase and position info from TIMER_A_IH.FW_PHASE
    and MECHANISM.FW_POSITION and store in packet.

    Get dichroic info from TIMER_A_IH.DM_PHASE and MECHANISM.DM_POSITION
    and and store in packet.

    Get TMPSU Secondary Currents from TMPSU.CURRENT and store in packet.

    Get status of ICB from ICB.STATUS and store in packet.

    Get SSI I/F error count from SSI_DRIVER.ERROR_COUNT
    and store in packet.

    Get Timing status's from TIME_MAN.SYNCHRONISATION_ACTIVE
    and TIME_MAN.VERIFICATION_ACTIVE
    and store in packet.

    Get RBI Status's from RBI.STATUS_REGISTER and RBI.CONFIG_REGISTER
    and store in packet.

    Get ICB Error Count from ICB.ERROR_COUNT and store in packet.

    Get TC Good Packet Counter from HK.TC_GOOD and store in packet.

    Get TC Bad Packet Counter from HK.TC_BAD and store in packet.

    Get OM Mode from MODEMAN.MODE and store in packet.

    Set ICU State to operational (=1) and store in packet.

    Get Which chain from value stored in ROM (i.e Prime or Redundant)
    and store in packet.

    Get S/W Version from value stored in ROM
    and store in packet.

    Get DPU Info from the DPU package and store in packet.

    then set the HK Packet SID field accordingly

    Get the current time and store in packet.

    Indicate CRC present

    Calculate and set the packet length field in the packet.

    Provided one of the 2 possible SID's are enabled

      Send packet to telemetry queue

  Check whether currently enabled HK SID has changed
  using TM_MAN.SID_STATUS.

  Calculate the next HK sample time
```

```
        (derived by adding last start of acquition time
        to the time interval between packets implied by the SID).

        Subtract it from the current time and delay the
        code by the result, thus ensuring the average time interval
        between HK packets is the expected time interval.

   end of infinite loop

procedure OFF(HK_WAS_RUNNING : out BOOLEAN) is


   Disable the HK acquisition program by calling the PROCESS.OFF entry point.

procedure ON is


   Ensure HK program is running by calling the PROCESS.ON entry point.

procedure BLOCK(ACTION : BOOLEAN) is


   Block HK by setting an appropriate flag.
```

#### *6.4.2.23 icb.ads*

Extracted from file "icb.ads"

```
    Function
    ========

    This file contains the specification for the ICB package. The package
    controls access to lower-level routines that interface directly with
    the Instrument Control Bus (ICB). The ICB is implemented using the
    MACSbus protocol.

--------------
package ICB is
--------------


      Define SUBADDRESS_TYPE


  procedure PUT(DEST      : DEST_ADDRESS_TYPE;
                SUBADR    : SUB_ADDRESS_TYPE;
                DATUM     : UINT16;
                OK        : out BOOLEAN);


      Writes DATUM to sub-address SUBADR at MACSbus destination DEST.

      Returns OK = TRUE if no errors occur.

  procedure GET(DEST      : DEST_ADDRESS_TYPE;
                SUBADR    : SUB_ADDRESS_TYPE;
                DATUM     : out UINT16;
                OK        : out BOOLEAN);


      Reads DATUM from sub-address SUBADR at MACSbus destination DEST.

      Returns OK = TRUE if no errors occur.

  procedure RESET;


      Resets the ICB MACSbus interface.

  function REPORT(TID : UBYTE;
                  FID : UBYTE)
                  return BOOLEAN;


      The function implements the "Read ICB Address Directly" command
      as described in section 2.2.5 of the Telecommand and Telemetry
      Specification, XMM-OM/MSSL/ML/0010.

      Specifically, it constructs a Task Parameter Report [TM(5,4)] containing
      the datum read back from subaddress FID at destination TID-40(hex), as
      documented in section 3.5 of the above document.

      In this release, it always returns TRUE.

  function STATUS
    return UBYTE
    renames ICB_DRIVER.HK_STATUS;


      For convenience, renames a low-level routine which returns
      the ICB interface status word - see package ICB_DRIVER for
      more details.

  function ERROR_COUNT
    return UBYTE
    renames  ICB_DRIVER.ERROR_COUNT;


      Returns the ICB error count (modulo 256) since the ICU was started.

   function BUSY return BOOLEAN;


       Returns TRUE if the ICB interface is being used by other code.
```

#### *6.4.2.24 icb.adb*

Extracted from file "icb.adb"

```
 Function
 ========

 This file contains the body for the ICB package. The package
 controls access to lower-level routines that interface directly with
 the Instrument Control Bus (ICB). The ICB is implemented using the
 MACSbus protocol.

-------------------
package body ICB is
-------------------


    The following procedures are internal to this package.

   procedure SEIZE;
   procedure RELEASE;


         SEIZE does not exit until it has seized the ICB interface
         for exclusive use.

         RELEASE release the ICB interface for use by other code.

    N.B. As the ICB interface code might be
         called at interrupt level, the required semaphore mechanism
         is implemented using critical sections
         (which are valid at interrupt level) in these procedures
         whilst manipulating a BUSY flag.
         The alternative of using the MUTEX package is not valid
         at interrupt level as it uses ADA tasking.

    Specify a default BUSY flag status of FALSE.

   procedure  RESET is


       If we are not already at interrupt level (failsafe test)

          Ensure that this routine has exclusive use of the MACSbus interface
          using SEIZE.

       Call the ICB driver low level reset function

       If we are not already at interrupt level (failsafe test)

          Release the MACSbus interface for use by other code using RELEASE.

   procedure PUT(DEST         : DEST_ADDRESS_TYPE;
                 SUBADR       : SUB_ADDRESS_TYPE;
                 DATUM        : UINT16;
                 OK           : out BOOLEAN) is


       If we are not already at interrupt level (failsafe test)

          Ensure that this routine has exclusive use of the MACSbus interface
          using SEIZE.

       Send the datum to the low level ICB PUT routine

       If we are not already at interrupt level (failsafe test)

          Release the MACSbus interface for use by other code using RELEASE.

   procedure GET(DEST         : DEST_ADDRESS_TYPE;
                 SUBADR       : SUB_ADDRESS_TYPE;
                 DATUM        : out UINT16;
                 OK           : out BOOLEAN) is


       If we are not already at interrupt level (failsafe test)

          Ensure that this routine has exclusive use of the MACSbus interface
          using SEIZE.

       Obtain a datum via the ICB low level driver GET function
```

```
    If we are not already at interrupt level (failsafe test)

       Release the MACSbus interface for use by other code using RELEASE.

function REPORT(TID          : UBYTE;
                FID          : UBYTE) return BOOLEAN is


   Get the datum at the address and sub-address corresponding
   with the supplied TID and FID.

   Supply the datum to the TASK_REPORT package to construct
   and send the aappropriate Report Task Parameters Packet.

   Return Success.

procedure SEIZE is


    Begin infinite loop

       Enter critical section

       If the BUSY flag is set

          Leave critical section

       Otherwise

          Set BUSY flag

          Leave critical section.

          Exit procedure.

       Wait a bit

       Then try again.

procedure RELEASE is


    Enter Critical Section.

    Set the BUSY flag to false.

    Leave Critical Section.

function BUSY return BOOLEAN is


   Return status of BUSY flag.
```

#### *6.4.2.25 icb_driver.ads*

```
Extracted from file "icb_driver.ads"


    Function
    ========

    This file contains the specification for the ICB_DRIVER package.
    The package provides the lower-level routines that interface directly
    with the Instrument Control Bus (ICB). The ICB is implemented using the
    MACSbus protocol.

---------------------
package ICB_DRIVER is
---------------------


  procedure PUT(DEST    : DEST_ADDRESS_TYPE;
                SUBADR  : SUBADR_ADDRESS_TYPE;
                DATUM   : UINT16;
                OK      : out BOOLEAN);


      This procedure write the datum DATUM to sub-address SUBADR at
      MACSbus destination DEST. OK is set to TRUE if no errors occur.

  procedure GET(DEST    : DEST_ADDRESS_TYPE;
                SUBADR  : SUBADR_ADDRESS_TYPE;
                DATUM   : out UINT16;
                OK      : out BOOLEAN);


      This procedure gets the datum DATUM from sub-address SUBADR at
      MACSbus destination DEST. OK is set to TRUE if no errors occur.

  procedure RESET;


      This procedure resets the MACSbus interface.

  function HK_STATUS return UBYTE;


      This procedure returns the status word of the ICB MACSbus interface
      BUT only for the last occurring error.

  function ERROR_COUNT return UBYTE;


      This returns the (modulo 256) error count of MACSbus errors since
      the ICU code started running.

      Provide a flag to be set when ICB_DRIVER is being called at interrupt level
      but default it to FALSE.
```

#### 6.4.2.26 icb_driver.adb

```
Extracted from file "icb_driver.adb"


 Function
 ========

 This file contains the body for the ICB_DRIVER package.
 The package provides the lower-level routines that interface directly
 with the Instrument Control Bus (ICB). The ICB is implemented using the
 MACSbus protocol.

 Dependencies
 ============

--------------------------
package body ICB_DRIVER is
--------------------------


   NOTE: The structure of the status register is as follows:

   --------------------------------------------------------
   | msb  |      |      |      |      |      |      | lsb  |
   ---------------------------------------------------------
   |   8  |   9  |  10  |  11  |  12  |  13  |  14  |  15  |
   ---------------------------------------------------------
   |    DEAD BITS              |      | TX   | EXT  | SYNC |  END |
   |                           |      | ERR  | ERR  |  ERR | COMM |
   --------------------------------------------------------

   Note: the structure of the ICB command register is:

   ---------------------------------
   | MSB                        LSB|
   ---------------------------------
   | ext  |   dest    |  subadr  |inst |
   |      |           |          |     |
   | | |  | | | | |   | | | | |  | | | |
   ---------------------------------

  function GET_STATUS return ICB_STATUS_TYPE is


     Read the ICB MACSbus status register port.

     Extract and return the status word

  function HK_STATUS return UBYTE is


     Return the last noted status word ** at the last error **.

  procedure PUT(DEST    : DEST_ADDRESS_TYPE;
                SUBADR  : SUBADR_ADDRESS_TYPE;
                DATUM   : UINT16;
                OK      : out BOOLEAN) is


     Construct command word to be written to command register
     based on supplied DEST and SUBADR
     (Note, Instr = RD = 010 binary, Ext = 101 binary)

     Write Datum to datum register port

     Write command word to command register (thus initiating transfer)

     Poll status word using GET_STATUS and then
     wait for completion of command (END COMM bit set),
     an error (i.e. TX ERR, EXT ERR or SYNC ERR bit set) or a timout, and
     remember the resulting status.

     Flag an error if any error bit was set , a timout or all 'dead bits' set.
     Otherwise, assume OK.

     If no error

       Do nothing.

     Otherwise
```

```
    Hand status, command word and datum over to be
    processed by the ANALYSE_ERRORS procedure.

  Finally, ensure interface is reset prior to next operation by
  calling procedure RESET

procedure GET(DEST    : DEST_ADDRESS_TYPE;
              SUBADR  : SUBADR_ADDRESS_TYPE;
              DATUM   : out UINT16;
              OK      : out BOOLEAN) is


  Construct command word to be written to command register
  based on supplied DEST and SUBADR
  (Note, Instr = TI = 100 binary, Ext = 101 binary)

  Write command word to command register port
  (which initiates transfer).

  Poll status word using GET_STATUS and then
  wait for completion of command (END COMM bit set),
  an error (i.e. TX ERR, EXT ERR or SYNC ERR bit set) or a timout, and
  remember the resulting status.

  Flag an error if error bit set or a timout or all 'dead' bits set.
  Otherwise assume OK.

  Get datum (this will be bad data if there was an error )

  If no error

    Do nothing.

  Otherwise

    Hand status, command word and datum over to be
    processed by the AANALYSE_ERRORS procedure.

  Finally, ensure status register is reset prior to next operation by
  calling procedure RESET.

procedure RESET is


  Reset the ICB interface by writing a "don't care" bit (i.e. any)
  pattern to the Status Register Port

  Note new status.

procedure ANALYSE_ERRORS(COMMAND_WORD : UINT16;
                         DATUM : UINT16;
                         STATUS : ICB_STATUS_TYPE) is


  Remember this error status for reporting by HK_STATUS.

  Increment the error count (modulo 256)

  Construct the appropriate 'MACSbus Error' Exception Report.

 Provided the 'at interrupt level' flag is not set

    send the appropriate 'MACSbus Error' Exception Report.

function ERROR_COUNT return UBYTE is


  Return the (modulo 256) error count.
```

#### *6.4.2.27 icu_mem_manager.ads*

Extracted from file "icu_mem_manager.ads"

```
    Dependencies
    ============

with TYPES; use TYPES;
with SYSTEM;


--------------------------
package ICU_MEM_MANAGER is
--------------------------


   function LOAD_MEMORY(MID: UINT16;
                        START_ADDRESS: LONG_INTEGER;
                        DATA: UINT16_ARRAY;
                        LENGTH: UINT16;
                        SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN;


       where MID is the MID
       where START_ADDRESS is the start address of the load
       where DATA is the data to load as an array of unsigned 16 bit words
       where LENGTH is the length of the data in words
       where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
       returns a boolean: true on success and false on failure
       function LOAD_MEMORY loads memory corresponding to the MID

   function DUMP_MEMORY(MID: UINT16;
                        ADDRESS: LONG_INTEGER;
                        LENGTH: UINT16;
                        SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN;


       where MID is the MID
       where ADDRESS is the address of the dump request
       where LENGTH is the length of the requested memory dump in words
       where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the
       sequence count and source
       returns a boolean: true on success and false on failure
       function DUMP_MEMORY dumps memory corresponding to the MID

   function CALCULATE_MEMORY_CHECKSUM(MID: UINT16;
                                      ADDRESS:LONG_INTEGER;
                                      LENGTH: UINT16;
                                      SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN;


       where MID is the MID
       where ADDRESS is the address of the crc request
       where LENGTH is the length of the requested block of memory to crc in words
       where SEQUENCE_COUNT_AND_SOURCE is a 16 bit word containing the sequence count and source
       returns a boolean: true on success and false on failure
       function CALCULATE_MEMORY_CHECKSUM calculates the checksum of the memory
       region corresponding to the MID
```

#### *6.4.2.28 icu_mem_manager.adb*

```
Extracted from file "icu_mem_manager.adb"


 Dependencies
 ============

with UNCHECKED_CONVERSION;
with ARTCLIENT;
with SYSTEM;
with INTRINSICS;

with MEMLOC;
with TYPES; use TYPES;
with PACKET;
with TC_VERIFY;
with TMQ;
with PEEK_POKE;
with CRC;
with DEBUG;
with TIME_MAN;
with NHK;
with ICB;


-------------------------------
package body ICU_MEM_MANAGER is
-------------------------------


   task MEMORY_DUMP is


   procedure SEND_PACKET(SUB_TYPE: PACKET.TELEMETRY_SUBTYPE; ADDRESS: LONG_INTEGER; DATA :
UINT16_ARRAY; LENGTH : UINT16; MID: UINT16) is


       Flag CRC as present

       Check if CRC is present

       If subtype is for a memory_dump

           Write the address into the packet

          Write the packet_length into the packet

          Write the data into the packet

          If subtype is for a memory_checksum_report

           Write the address into the packet

          Write the packet_length into the packet

          Write the memory_length into the packet

       Send the packet

   procedure READ_BLOCK(MID: UINT16; ADDRESS: LONG_INTEGER; LENGTH: INTEGER; DATA: in out
UINT16_ARRAY; SEQUENCE_COUNT_AND_SOURCE: UINT16) is


       returns array 0 .. PACKET.MAX_TM_MEM_PARAMS_M1

       Check the MID

             Check whether we want ICU, Window Bitmap Table or Centroid Lookup Table

                When the MID is 0: icu operand/data space
                For each word of data to be read

                   Calculate the address state

                   Enter critical section

                   Read from the address

                   Leave critical section

                Read status
```

```
            If not accessible by ICU

            make it so

            Set the start address
            Be careful: only least sig 8 bits autoincrement

               Send the address again if the least sig 8 bits are 0

            Restore status

            Read status

            If not accessible by the ICU

               make it so
               If not accessible enable for ICU access

            Set the start address
            The 16 bits autoincrement

            Finally, disable for ICU access

         When the MID is 1: icu instr space
         For each word of data

            Calculate the address_state

            Enter critical section

            Read from the address

            Leave critical section

         When the MID is wrong
         Send unsuccessful acceptance packet

   task body MEMORY_DUMP is


       begin an infinite loop

          if a call to start is made

            Finish when there's nothing left

            If there's more than a packet left

               Read the memory

               Send the data in a packet

               Recalculate the no of words left

               If there's less than or just one packet left
               Read the memory

               Send the data in a packet

   function LOAD_MEMORY(MID: UINT16; START_ADDRESS: LONG_INTEGER; DATA: UINT16_ARRAY; LENGTH:
UINT16; SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


         When the MID is 0: icu operand/data space
         For each word to be loaded

            if address is in the interrupt vector table - don't write it

               Calculate address state and address offset

               Enter critical section to
               protect from address state change

               Write

               Leave critical section

         When the MID is 1: icu instruction space
         For each word to be loaded

               Calculate address state and address offset
```

```
                  Protect from address state change by entering critical section

                  Write the value to memory

                  Leave critical section

            Otherwise the MID must be wrong
            put params in array

            Send unsiccessful acceptance (illegal mid) packet

   function DUMP_MEMORY(MID: UINT16; ADDRESS: LONG_INTEGER; LENGTH: UINT16;
SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


       Remember the dump parameters

       Try to ask for dump

          for 0.5 second

          if can't dump, return false so that an unsuccessful execution can be sent

   function CALCULATE_MEMORY_CHECKSUM(MID: UINT16;
                                      ADDRESS: LONG_INTEGER;
                                      LENGTH: UINT16;
                                      SEQUENCE_COUNT_AND_SOURCE: UINT16) return BOOLEAN is


       Set crc syndrome to ffff to start with

       loop

          until there's nothing left to crc

          If there's more than or just one packet's worth left

             Read a block of memory

             crc it

             recalculate length remaining

             If there's less than a packet's worth left
             Read a block of memory

             crc it

             finish

       Send a memory checksum report with the checksum just calculated
```

#### *6.4.2.29 importance.ads*

Extracted from file "importance.ads"

```
    Function
    ========

    This file contains the specification only package IMPORTANCE.
    This package defines the priority of tasks

    The range of priorities is 10..200
    The default is SYSTEM.DEFAULT_PRIORITY := 10;

    Priorities are allocated in bands as follows:-

    H/W Simulators (for debugging)              191 -> 200
    RBI Watchdog reset                          190
    S/W Watchdogs                               171 -> 189
    "Semaphore" Tasks                           131 -> 140
    "Monitor Tasks" (eg. DPU, TM)               111 -> 130
    "Working Tasks" e.g. HK, Science, Blue      11  -> 110
    "Idle" Task                                 10

---------------------
package IMPORTANCE is
---------------------


    Priority Definitions
    ====================

    CPU Watchdog Reset
    ------------------

  CPU_RESET             : constant SYSTEM.PRIORITY := 190;


    Software Watchdogs
    ------------------

    DPU Heartbeat Watchdog Task

  DPU_HEARTBEAT         : constant SYSTEM.PRIORITY := 171;


    "Semaphore" Tasks
    -----------------

      Priority of Mutual exclusion semaphore task type

  MUTEX_SEMAPHORE       : constant SYSTEM.PRIORITY := 132;


      Timer A Resource

  TIMER_A               : constant SYSTEM.PRIORITY := 133;


    "Monitor Tasks" (eg. DPU, TC)
    -----------------------------

      Priority of Task to monitor DPU data for events

  DPU_DATA_MANAGER      : constant SYSTEM.PRIORITY := 111;


      Priority of Task to monitor Telecommand queue

  TCPROC                : constant SYSTEM.PRIORITY := 113;


  SAFING                : constant SYSTEM.PRIORITY := 112;


    "Working Tasks" (e.g. HK, Science, Blue)
    ----------------------------------------

    Load Blue Centroid Table

  LOAD_CENTROID_TABLE   : constant SYSTEM.PRIORITY := 91;
```

```
  Load Blue Window Table

LOAD_WINDOW_TABLE     : constant SYSTEM.PRIORITY := 92;

    Priority of task that collects and send HK data

HK_PROCESS            : constant SYSTEM.PRIORITY := 93;

  HV ramp task

HV_RAMP_TASK          : constant SYSTEM.PRIORITY := 94;

    Priority of task to perform Thermal Control

THERMAL_CONTROL       : constant SYSTEM.PRIORITY := 95;
```

#### *6.4.2.30 INTVEC.asm*

**File is INTVEC.asm**

```
;         Interrupt Vectors
;
;    This file defines the statically initialized interrupt vectors
; for the Tartan runtimes.  It also defines the starting address of the
; program image.   Users may wish to add interrupt vector definitions or
; modify the startup sequence as their applications evolve.  NOTE: when
; using TLC or Adascope, unused interrupt vectors may be uninitialized;
; the debug kernel will intercept such unused interrupts.
;
; TAKE CARE to set the following configuration flags properly!
;*********************************************************************
EXPANDED_MEM   EQU  0    ; ONE => Set up for expanded memory runtimes
DEBUG_VERSION  EQU  1    ; ONE => Set up for use with debug kernel
TASM EQU  0         ; ONE => Tartan Assembler (do not set)
; end of configuration flags

     REFER     NUMERIC_O_LP   ; integer overflow linkage ptr
     REFER     NUMERIC_O_SP   ; integer overflow service ptr
     REFER     TIMER_B_LP     ; timer B linkage ptr
     REFER     TIMER_B_SP     ; timer B service ptr
     REFER     ADAROOT        ; starting point of Ada runtimes
     REFER     BCP4_LP
     REFER     BCP4_SP
     REFER     SSI_LP
     REFER     SSI_SP
     REFER     RBI_LP
     REFER     RBI_SP
  IF EXPANDED_MEM   ;!!!!
     REFER     BEX_STATE ; "branch to executive" linkage ptr
     REFER     BEX_TABLE ; "branch to executive" service ptr
  ENDIF   ;!!!!

     ABSOLUTE

;*********************************************************************
;
; The Ada runtime startup is at ADAROOT.  How it is started depends upon
; the boot sequence for your system.  Bare hardware starts up at 0,
; the debug kernel obeys the specified starting address.  Expanded
; memory with the Tartan toolset uses a more careful init sequence.
;
;*********************************************************************
  IF DEBUG_VERSION  ;!!!!!
; kernel uses power-up vector
  ELSE    ;!!!!
; start by power-up sequence, jump to initialization code
     ORIGIN    0
     JC   7,INIT_RT
  ENDIF   ;!!!!

  IF EXPANDED_MEM   ;!!!!
; see exciting init code at the end of the file
  ELSE    ;!!!!
; debug kernel starts us, just avoid overwriting his vectors
     ORIGIN    01E
INIT_RT   JC   7,ADAROOT ; jump to real start addr
  ENDIF   ;!!!!

;*********************************************************************
;
; MIL-STD-1750 Interrupt vectors.  Only those needed by a debug version
; are initialized below.
;
;*********************************************************************
     ORIGIN    020       ; MIL-STD-1750 start of vectors
     DEFINE    ART1750VEC     ; runtimes refer by this name
ART1750VEC     EQU  $
;    DATA ?,?                 ; (0) Power Down
;    DATA ?,?                 ; (1) Machine Error
;    DATA ?,?                 ; (2) Spare
     ORIGIN    026
     DATA NUMERIC_O_LP,NUMERIC_O_SP     ; (3) Floating point overlow
     DATA NUMERIC_O_LP,NUMERIC_O_SP     ; (4) Fixed point overflow
  IF EXPANDED_MEM & (DEBUG_VERSION==0)  ;!!!!
     DATA BEX_STATE,BEX_TABLE      ; (5) BEX
  ENDIF   ;!!!!!
;    DATA ?,?                 ; (6) Floating point underflow
```

```
;     DATA ?,?                 ; (7) TIMER A
      ORIGIN    030
      DATA BCP4_LP,BCP4_SP               ; (8) BCP4
      ORIGIN    032
   IF TASM ;!!!!
      DATA WEAK$TIMER_B_LP,WEAK$TIMER_B_SP    ; (9) TIMER B
   ELSE    ;!!!!
      DATA TIMER_B_LP,TIMER_B_SP         ; (9) TIMER B
   ENDIF   ;!!!!
      ORIGIN    034
      DATA SSI_LP,SSI_SP            ; (10) SSI interrupt
;     data ?,?                 ; (11) Spare
;     DATA ?,?                 ; (12) IN/OUT 1
      ORIGIN  03a
      DATA RBI_LP,RBI_SP            ; (13) RBI interrupt
;     DATA ?,?                 ; (14) IN/OUT 2
;     DATA ?,?                 ; (15) Spare

;*************************************************************************
;
; Program startup in expanded memory is more interesting because the
; world comes up in an unmapped state, but the image is linked to run
; in a mapped environment.  Thus we must (carefully) at startup initialize
; the page registers.  The code below solves this problem.  The placement
; is selected to avoid the debug kernel.
;
;*************************************************************************
   IF EXPANDED_MEM   ;!!!!
      REFER     SEGMENT$TABLE  ; page table built by the linker
AS1REGS   EQU  010       ; offset for AS1 page registers
R0   EQU  0
R1   EQU  1
R2   EQU  2

      ORIGIN    0240
; We are started here by the debug kernel, or power-up.
; We assume that virtual I and D page 0 point to this code.
INIT_RT   XIO  R0,RIPR+0 ; get mapping for this page (ASSUMES VIRT 0!)
      XIO  R0,WIPR+AS1REGS     ; init AS1 I page 0 to point here
      XIO  R0,WOPR+AS1REGS     ; init AS1 D page 0 to point here
      LISP R2,1      ; AS1
      XIO  R2,WSW         ; now we are executing in AS1
      DL   R0,ART_SEGLOC  ; get PHYSICAL address of segment table
      DSLL R0,4      ; move page number bits to R0
      XIO  R0,WOPR+AS1REGS+15  ; set into AS1 D page 15
      SRL  R1,4      ; rejustify page offset
      ORIM R1,0F000  ; page offset in page 15
      VIO  R2,0,R1        ; load up AS0 I pages
      VIO  R2,18,R1  ; load up AS0 D pages
      LST  TOADA          ; go back to AS0 and ADAROOT
; associated data
ART_SEGLOC     EQU  $
      PHYSICAL SEGMENT$TABLE
TOADA     DATA 0    ; mask
      LOGICAL   ADAROOT   ; sw, ic (ADAROOT must be in seg 0)
   ENDIF   ;!!!!
      END  INIT_RT
```

#### 6.4.2.31 mechanism.ads

Extracted from file "mechanism.ads"

```
    Function
    ========

    This file contains the specification for the MECHANISM package. This
    represents the Filter Wheel and Dichroic mechanism objects

------------------
package MECHANISM is
------------------


  function MOVE_FILTER_WHEEL(SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN;


      Instructs filter wheel to perform the movement specified
      by SET_FILTER_WHEEL_MOVEMENT

      where :

      SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
      of the associated telecommand.

      Returns TRUE if command was successfully accepted


  function SET_FILTER_WHEEL_MOVEMENT(FW_MOVEMENT            : FW_MOVEMENT_TYPE;
                                     VALUE                  : UINT16;
                                     SRC_AND_SEQUENCE_COUNT : UINT16)
                                     return BOOLEAN;


      Informs package of what type of filter wheel movement is to be performed
      by the next call to MOVE_FILTER_WHEEL.

      where :

      FW_MOVEMENT specifies the type of filter wheel movement required.
       4 : To the filter number (0 -> 11) given by VALUE
       5 : To the absolute position given by VALUE (0->2199 steps from datum)
       6 : To the relative number of steps from the current one
       7 : To VALUE number of fine sensor pulses
       8 : To the Datum position
       9 : To the first sensing of the coarse sensor.

      VALUE specifies any numerical value (e.g. how many steps) associated
      with the type of movement (only examined if relevant)

      SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
      of the associated telecommand.

      Returns TRUE if command was successfully accepted

  function SET_DICHROIC_DIRECTION(DIRECTION : INTEGER;
                                  METHOD    : UINT16;
                                  SRC_AND_SEQUENCE_COUNT : UINT16)
                                  return BOOLEAN;


      Informs the package of the direction and type of dichroic motion
      to be executed on the next call to MOVE_DICHROIC.

      where :

      DIRECTION specifies the direction (-ve = Redundant to Primary, +ve
      Primary to Redundant) and, in the case of METHOD = 1, the number of
      steps the dichroic is to move.

      METHOD specifies the type of dichroic movement required:
        0 = Dichroic is moved to its maximum excursion in the direction
            indicated by the sign of DIRECTION
        1 = Dichroic is moved by the magnitude of DIRECTION in the direction
            indicated by the sign of DIRECTION

      SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
      of the associated telecommand.
```

```
    Returns TRUE if command was successfully accepted

function MOVE_DICHROIC(SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN;


    Requests the dichroic to move as specified by the priorn call to
    SET_DICHROIC_DIRECTION

    where:

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if command was successfully accepted

function CHANGE_FW_STEP_RATE(PULL_IN_RATE : UINT16;
                             CRUISE_RATE  : UINT16;
                             ACCELERATION : UINT16;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN;


    Changes the acceleration parameters for the filter wheel.

    where:

    PULL_IN_RATE is the startup pulse rate (hz)

    CRUISE_RATE is the maximum pulse rate (hz)

    ACCELERATION is the acceleration used to go from PULL_IN_RATE
    to CRUISE_RATE (hz/sec)

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if command was successfully accepted

function CHANGE_DICHROIC_STEP_RATE(NEW_RATE : UINT16;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN;



    Changes the acceleration parameters for the dichroic.

    where:

    NEW_RATE is the new step rate (hz)

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if command was successfully accepted

function STOP_FILTER_WHEEL(SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN;


    Stops the filter movement (if active).

    where:

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if command was successfully accepted

function FW_POSITION return UINT16;


    Returns the current fw position for HK display

    0 -> 2199 : Number of steps from datum
    2200      : Filter Wheel position unknown
    2201      : Filter Wheel Moving

function LAST_FW_MOVEMENT_OK return INTEGER;


    Returns result of last f/w movement

    -1 : Still Moving
```

```
     0 : Unsuccessful
     1 : Successful

function DM_POSITION return INTEGER;


    Returns the current dichroic position for HK display

    -32 -> 31 : Number of steps from position at start of operational mode
                (-ve : toward Primary; +ve : Towards Redundant)

function AT_COARSE_SENSOR return BOOLEAN;


    Returns TRUE if filter wheel coarse sensor was detected when
    last examined.

function AT_FINE_SENSOR return BOOLEAN;


    Returns TRUE if filter wheel fine sensor was detected when
    last examined.

procedure INIT;


    Initialises the mechanisms package

procedure AWAIT_DPU_HEARTBEAT;


    This procedure is a rendevous point. It is called by the DPU package to
    inform the mechanism package that a DPU heartbeat has been received.
    It times out after 11 secs.

function PERFORM_FW_SAFING(SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN;


    Request the Filter Wheel to move to a 'Safe' position.

    where:

    SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
    of the associated telecommand.

    Returns TRUE if command was successfully accepted

    The block of variables are now declared as part of the specification
    so that they are 'visible' to the TIMER_A_IH package which actually
    performs the movement. That package is compiled separately as it is run
    at interrupt level and therefore a different set of compilation flags
    must be used.
```

#### 6.4.2.32 mechanism.adb

Extracted from file "mechanism.adb"

```
 Function
 ========

 This file contains the body for the MECHANISM package. This
 represents the Filter Wheel and Dichroic mechanism objects


-------------------------
package body MECHANISM is
   -------------------------


    The following are specifications for functions, procedures and
    tasks internal to the package.

   procedure TERMINATE_MOVEMENT;


   function CHANGE_PULSE_RATE(DEVICE       : in DEVICE_TYPE;
                              PULL_IN_RATE : in UINT16;
                              CRUISE_RATE  : in UINT16;
                              ACCELERATION : in UINT16;
                              SRC_AND_SEQUENCE_COUNT : UINT16)
                              return BOOLEAN;


   task MECH is

      pragma priority(IMPORTANCE.TIMER_A);

      entry AWAIT_DPU_HEARTBEAT;


      entry ACTIVATE;


      entry DEACTIVATE;


   end MECH;


    where
    entry AWAIT_DPU_HEARTBEAT pauses the task until the next DPU heartbeat.
    entry ACTIVATE starts moving the specified mechanism
    entry DEACTIVATE aborts the mechanism movement

    Now commence descriptions of bodies.
    ====================================

   function SET_FILTER_WHEEL_MOVEMENT(FW_MOVEMENT            : FW_MOVEMENT_TYPE;
                                     VALUE                  : UINT16;
                                     SRC_AND_SEQUENCE_COUNT : UINT16)
                                     return BOOLEAN is


       Examine the requested filter wheel movement.

          If we are specifying a move to a filter position

             Provided we are not in safe mode

                Store the parameters
                Set up exit condition as 'after required steps commanded'

             Else

                Inform ground that this is not valid for this mode

                and return an error flag.

          If we are specifying a move to an absolute position

             Store the values
             Set up exit condition as 'after required steps commanded'

          If we are specifying a move to an relative position
```

```
         Store the values
         Set up exit condition as 'after required steps commanded'

      If we are specifying a move to fine sensor

         Store the values
         Set up exit condition as 'at next fine sensor detection'

      If we are specifying a move to datum

         Store the values
         Set up exit condition as 'at detection of coarse and fine sensor'

      If we are specifying a move to the coarse sensor

         Store the values
         Set up exit condition as 'at detection of coarse sensor'

      Otherwise

         Do nothing

   Remember which type of movement was requested in FW_MOVEMENT_REQUESTED.

   Return without error

function MOVE_FILTER_WHEEL(SRC_AND_SEQUENCE_COUNT : UINT16)
                           return BOOLEAN is


   First get current position

   Is the F/W moving

      If so, tell ground it is busy

      and exit as an error

   Now set up f/w move on the basis of movement type stored in FW_MOVEMENT_REQUESTED.

      If it's a move to a filter position

         and if we are in safe mode

            tell the ground that this is invalid

            Store in LAST_FW_MOVEMENT that the last f/w movement was invalid

            and return with an error condition.

         (Re)Set up focussing heaters for this filter

         and the sample time

         Set parameter allowing acceleration of filter wheel at start

         If the f/w current position is unknown
         (e.g. not been to datum yet)

            Tell the ground

            Remember that this f/w movement was invalid

            and return with an error condition

         Determine the final step position the requested filter corresponds to

         If we are already at the requested position

            Send message to ground signifying success

            Store in LAST_FW_MOVEMENT that this f/w movement was valid

            Re-inform DPU of position of f/w (part of NCR 166) via DPU.SET_FILTER

            and return without error

         Otherwise

            Determine how many steps have to be moved from current position

         Store in INFORM_DPU that we must interact with the DPU when moving
```

```
        Also determine if it is valid to check fine sensor for this
        filter after movement (i.e. if final position is a multiple of 200)

    If its a move to an absolute position

        Set parameter allowing acceleration of filter wheel at start

        If the f/w current position is unknown
        (e.g. not been to datum yet)

            Tell ground about it

            Store in LAST_FW_MOVEMENT that this f/w movement was invalid

            and return and error condition.

        If we are already at requested position

            Send message to ground signifying success

            Store in LAST_FW_MOVEMENT that this f/w movement was a success

            Return with no error

        Otherwise

            Determine how many steps are to be moved

    If we are moving a relative number of steps

        Set parameter allowing acceleration of filter wheel at start

    If we are moving to a fine sensor position

        Set parameter NOT allowing acceleration of filter wheel at start

        Ensure fine sensor is on via DETECTOR.FINE_SENSOR.

        and flag that it should be checked for visiblity after movement

    If we are moving to the datum position

        Set parameter NOT allowing acceleration of filter wheel at start

        Flag that we should check fine sensor after movement

        Ensure coarse and fine sensors are on using TMPSU.COARSE_SENSOR
        and DETECTOR.FINE_SENSOR

        Wait a short while to allow them to settle.

        Check whether we can already see both the coarse and
        fine sensors.

            If so, we are already at datum

            Ensure fine and coarse sensors are off.

            Tell ground we are successful

            Flag in LAST_FW_MOVEMENT that this f/w movement was successful

            Set f/w position to zero

            Return with no error

    If we are moving to coarse

        Set parameter NOT allowing acceleration of filter wheel at start

        Ensure coarse sensor on using TMPSU.COARSE_SENSOR

        Wait a bit to allow it to settle.

        Check whether we can already see the coarse sensor

        If we can

            Ensure coarse sensor off

            Tell ground we are successful
```

```
          Note in LAST_FW_MOVEMENT that this f/w movement as successful

          Return without error.

      Otherwise

   Set parameter indicating we are about to move the f/w mechanism

   Set Initial Phase Increment to 1

   Get number of step movements to perform obtained earlier

   Get when we must exit determined in SET_FILTER_WHEEL_MOVEMENT

   Determine if this is an autosafing internally generated command.

   Check whether the f/w has not completed any previous commanded movement

       If so, issue a 'busy' message to ground.

    Otherwise

       Activate the movement (but don't wait for completion)

          Attempt to start the f/w moving using MECH.ACTIVATE

          and return without error

          Or timout if the code is busy

          and tell ground it is busy.

          and return with error flag set.

   Return without error.

function MOVE_DICHROIC(
                       SRC_AND_SEQUENCE_COUNT : UINT16)
                       return BOOLEAN is

   If if not in engineering mode

      Inform ground the command is invalid for this mode

      and return an error condition

   Note when commanded movement should cease
   (i.e. at requested +ve or -ve excursion)

   Set parameter indicating non-accelerating motion (always so for dichroic)

   Zero dichroic position counter

   Set parameter indicating that we are about to move the dichroic mechanism

   Set up iphase increment on basis of movement direction (1 for +ve, 3 for -ve)

   Allow no more than 35 steps

   Activate the motion (but don't wait for completion) using MECH.ACTIVATE

   Return without error

task body MECH is

   Now commence main task body

   Ensure 31750 Timer A is stopped

   Begin infinite loop

      Await call to an 'accept' point

         allow acceptance of a activate request

        accept ACTIVATE do

            Inform the TIMER_A_IH package that we are now moving a mechanism
```
--+ Inform the TIMER_A_IH package that we are now moving a mechanism

```
      TIMER_A_IH.MOVEMENT_FINISHED := FALSE;

   end ACTIVATE;

    Look at which mechanism is being commanded

       If it's the filter wheel we are moving

          Set flag indicating that filter wheel may no longer be in a safe position

          Remember current f/w position before moving

          Stop any DPU science data handshake using DPU.ENABLE_REQ_DATA

          Await a DPU heartbeat (or timout after 11 secs )

          Stop HK

          Turn on the coase and sensors

          If the current f/w position is unknown

             assume we are at the start

          Calc current phase on basis of current position

          Determine when we must start braking
          (as a function of acceleration and peak motion)

          Assume as a default success completion flag

          If it's the dichroic we are moving

          (Set Initial phase value)

          If we are moving to the maximum excursion

             Assume initial phase to be 1

          If we are moving n steps

             Set to last value used

          Set braking distance to zero

    Set mechansisms code as 'in use'

    Disable heaters, if any are on, to minimise power

    And load/start timer A with an interpulse gap value
    appropriate to pull-in speed for given mechanism

    Then send command to start Timer A pulse train using TIMER_A_IH.START

 or allow acceptance of an abort request

   accept DEACTIVATE do


       Stop Timer A interrupts procedure via TIMER_A_IH.STOP

       Flag that we are aborting

       Examine which device is being commanded

          If it's the filter wheel

             Determine appropriate failure message to send to ground

             Set F/W position in HK as unknown

             Flag last f/w movement as unsuccessful

             If it's the dichroic

             Determine appropriate failure message to send to ground

  Or if mechanisms are in use

       Every 1/2 sec

       Check to see if the movement has finished using TIMER_A_IH.MOVEMENT_FINISHED
```

```
              And terminate the moevement cleanly using TERMINATE_MOVEMENT

  Define procedure internal to the mechanism control task that is
  called at the termination of any mechanism movement.

 procedure TERMINATE_MOVEMENT is


     Ensure Timer A of the 31750 chip is stopped using TIMER_A_IH.STOP.

     Ensure all phase lines are set off;

     Look at which mechanism is in use.

        If it's the filter wheel

           Remember that this movement was good.

          Wait a bit to allow mechanisms to settle

          Get fine and coarse sensor values for HK

          (Set up f/w position for HK)

          If it was previously flagged as unknown position in HK,
          and we have not performed a move to an known position

             Ensure it is still flagged as unknown in HK

          Otherwise

             Make new position visible to HK

          If it's a f/w movement to a filter or a fine sensor only,

             If we should check the fine sensor but it is not seen

                flag it and determine appropriate message

               Suppress any later success messages

                Remember this movement as unsuccesful

          If it was a move to datum

             If we can't see both fine and coarse sensors

                Set f/w position as unknown in HK

                Determine appropriate message to send to ground indicating failure

                Suppress any further success messages

                and remeber this last f/w movement as unsuccessful

           If flagged as appropriate, inform DPU of requested f/w filter
           position if all OK using DPU.SET_FILTER

          Turn off coarse and fine sensors

          Determine whether we should send success message to ground
          if not suppressed earlier

          Override any message if movement was aborted by ground

          Unblock HK

          Renable DPU science data handshakes (i.e. restart
          downloading data

        If it's was a Dichroic motion

           Determine message to send to ground

     Send out appropriate NHK message determined above

     Renable heaters if any

     If NHK_MESSAGE = FW_LOST_POSITION

        Issue command to go to safe internally
```

```
          If it was successful

            Send NHK anomaly message to ground saying so

          Otherwise

            Send NHK message to ground saying an auto-safing attempt failed

       Release mechanisms code for use

       Return from termination of movement procedure

function FW_POSITION return UINT16 is


    Return current value of f/w position counter

function DM_POSITION return INTEGER is


    Return current value of Dichroic Position counter

function  CHANGE_PULSE_RATE(DEVICE       : in DEVICE_TYPE;
                            PULL_IN_RATE : in UINT16;
                            CRUISE_RATE  : in UINT16;
                            ACCELERATION : in UINT16;
                            SRC_AND_SEQUENCE_COUNT : UINT16
                           ) return BOOLEAN is


    Store the new rate provided it's sensible

    Otherwise signal an error

    always return success

function CHANGE_FW_STEP_RATE(PULL_IN_RATE : UINT16;
                             CRUISE_RATE  : UINT16;
                             ACCELERATION : UINT16;
                             SRC_AND_SEQUENCE_COUNT : UINT16)
                             return BOOLEAN is


    Attempt to change f/w step rates using CHANGE_PULSE_RATE

function SET_DICHROIC_DIRECTION(DIRECTION : INTEGER;
                                METHOD    : UINT16;
                                SRC_AND_SEQUENCE_COUNT : UINT16)
                                return BOOLEAN is


    If we are not in engineering mode

       Inform ground of failure

       Return with an error condition

    Note which method of movement (step by step or to max excursion)
    and which direction

    Return success condition.

function CHANGE_DICHROIC_STEP_RATE(NEW_RATE : UINT16;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN is


    Attempt to change dichroic pulse rate

function STOP_FILTER_WHEEL(SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN is


       Attempt to stop the f/w moving

       Return TRUE if successful

       Or timout if the code is busy

       Send Ground a 'busy' message

       Return FALSE

procedure SEND_NHK_PACKET(NHK_SID : PACKET.SID_TYPE; CODE : UINT16) is
```

```
    Load up condition code into an NHK packet

    Determine whether it's an event or major anomaly on the
    basis of the SID

    Provided originally this was not an autosafing internally
    generated command.

       Place an NHK packet in the telemetry queue

function LAST_FW_MOVEMENT_OK return INTEGER is


   Return whether last f/w movement was successful

function AT_COARSE_SENSOR return BOOLEAN is


    Return whether the coarse sensor was seen when last checked

function AT_FINE_SENSOR return BOOLEAN is


    Return whether the fine sensor was seen when last checked

procedure INIT is


    Ensure coarse and fine sensors are on

    Wait a bit to let them settle

    Determine sensor status for HK

    Ensure coarse and fine sensors are off

procedure AWAIT_DPU_HEARTBEAT is


       Await a heartbeat from the DPU

function PERFORM_FW_SAFING(SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN is


    Set the coarse sensor current to 4

    Return FALSE if it fails

    Set the fine sensor current to 9

    Return FALSE if it fails

    If the current filter wheel is already safed

       Send message to ground signifying success

    else

       if the filter wheel position is already known

          Then command filter wheel to move to the blocked position (filter 0)
          Will not move the filter wheel if already at blocked

          else request the filter wheel to find the coarse position
          if not already at blocked

             Activate the filter wheel movement.

                Wait for the movement to complete

                If movement was good

                   Request the filter wheel to move 1258 steps from the coarse position.
                   This should make it move to the blocked position.

                      Activate the filter wheel movement.

    Record safing outcome
```

#### *6.4.2.33 mem_manager.ads*

Extracted from file "mem_manager.ads"

```
    Function
    ========

    This file contains the specification for package mem_manager.
    That package calls icu_mem_manager or dpu_mem_manager to load/dump/check memory.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/SP/0061
    Dependencies
    ============

with TYPES; use TYPES;
with PACKET;


   --------------------------------------------------------------------
   function REQUEST(MEM_MANAGER_PACKET: PACKET.TC_TYPE) return BOOLEAN;
   --------------------------------------------------------------------


      Where MEM_MANAGER_PACKET is a memory management packet
      Returns BOOLEAN true success or false on failure
      This merely forwards packets onto the ICU_MEM_MANAGER package or the
      DPU_MEM_MANAGER package
```

#### *6.4.2.34 mem_manager.adb*

Extracted from file "mem_manager.adb"

```
 Function
 ========

 This file contains the body for package mem_manager.
 It calls icu_mem_manager or dpu_mem_manager to load/dump/check memory.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/SP/0061

 Dependencies
 ============

with UNCHECKED_CONVERSION;

with PACKET;
with ICU_MEM_MANAGER;
with DPU_MEM_MANAGER;
with TMQ;
with TC_VERIFY;
with DEBUG;
with MODEMAN;
with NHK;


---------------------------
package body MEM_MANAGER is
---------------------------


   function REQUEST(MEM_MANAGER_PACKET: PACKET.TC_TYPE) return BOOLEAN is


       Find length of CRC (is it there or not)

       Calculate length of data in packet

       Convert length from bytes to words

          Check memory management packet subtype - load/dump/crc

          If it is a load command (subtype 1)

             Check the MID

                When the MID is for the DPU

                      Call LOAD_MEMORY in DPU_MEM_MANAGER

                   Otherwise send an unsuccessful acceptance packet

                Return FALSE if something went wrong

             When it's a dump memory command (subtype 2)

             If length is out of range, send an error packet

             Check the MID

                When the MID is for the ICU (0, 1)

                   Call DUMP_MEMORY in ICU_MEM_MANAGER

                      if we had trouble, send an unsuccessful execution packet

                   When the MID is for the DPU (10h-15h, 20h-27h)

                   Call DUMP_MEMORY in DPU_MEM_MANAGER

                   Otherwise send an unsuccessful acceptance packet

             When it's a memory crc (subtype 3)
             Check the length

             Check the MID
```

```
   If the MID is for the ICU (0, 1)

      Call CALCULATE_MEMORY_CHECKSUM in ICU_MEM_MANAGER

      Otherwise send an unsuccessful acceptance packet

Otherwise we have a wrong subtype for MEM_MANAGEMENT
So send an unsuccessful acceptance
```

#### *6.4.2.35 memdpu.ads*

Extracted from file "memdpu.ads"

```
    Function
    ========

    This file contains the specification for the package MEMDPU. That package
    constructs Memory Dump packets from DPU RAM dump blocks (i.e. blocks
    of the type DR_xxx) and places them in the telemetry queue. The format of
    the DR_xxx blocks are defined in section 6 of the 'XMM-OM ICU-DPU Protocol
    Definitions', XMM-OM/MSSL/ML/11.

-----------------------
package  MEMDPU is
-----------------------


  procedure PUT(DPU_DATA        : UINT16_ARRAY) ;


      This procedure constructs Memory Dump packets from the supplied
      DPU DR_xxx block contained in DPU_DATA. Packets deemed complete (i.e.
      when they are the maximum length that can be accomodated for that
      particular type of data) are then sent to the telemetry queue..

      NOTE: the index of this array must start at 0.

  procedure FLUSH;


      This procedure causes any memory dump packets not occupying the maximum
      length to be flagged as complete and sent on to the telemetry queue.
```

#### 6.4.2.36 memdpu.adb

Extracted from file "memdpu.adb"

```
 Function
 ========

 This file contains the body for the package MEMDPU. That package
 constructs Memory Dump packets from DPU RAM dump blocks (i.e. blocks
 of the type DR_xxx) and places them in the telemetry queue. The format of
 the DR_xxx blocks are defined in section 6 of the 'XMM-OM ICU-DPU Protocol
 Definitions', XMM-OM/MSSL/ML/11.

-----------------------
package body MEMDPU is
-----------------------


      Declare an Instance of the Packet Record

   The following specification is for a procedure internal to the package.

  procedure OUTPUT_DPUMEM;


   Adds header to memory dump packet and sends it to the telemtry queue.

  procedure PUT(DPU_DATA        : UINT16_ARRAY) is


     Assume, by default, the data should be 'packed' into
     the packet (see below).

     Set up default location of where to copy data from in the DPU block.

     Get the DPU DR_xxx block type.

     Extract starting address of DPU RAM data from the DPU block.

     Calc default number of words to copy from DPU block into packet(s).

     If it's a DR_LRM block (a dump from local ram)

       For this particular type of DR_xxx data
       1) Correct how many words to copy from the DPU block
       2) Correct where to copy the words from the block (the 'base')

       Because of larger internal header, decrease words to copy by 6.

       Set the MID

       Extract the DPU local memory address for the start of data.

       Derive the 'base'

     Else, if its a DR_PROG_DUMP block (Dump of program RAM)

       Derive the MID as a function of the start address
       contained in the block.

     Else, if it's a Global Ram Dump (DR_RAM_DUMP or DR_RAM_DUMP_N_ZERO)

       If the start address is in small word memory.

         Specify the MID accordingly

         And flag that the data should not by 'packed'

       Otherwise

         Specify the MID accordingly.

     Loop over data to be copied from the DPU block,
     starting at 'base' derived above.

       If we are at the start of a packet

         Store, in the packet, the DPU memory address corresponding
         to the DPU words also about to be copied into the packet.

       Copy data into work area one word at a time
```

```
   (as data may span DPU blocks we need to keep a copy so we can
   join the next block to this one)

   Increment how many words we have copied into work area

   If we are to 'pack' the words into the packet

     If we have accumulated 4 words since the last packing operation, pack again

       The DPU words are 24 bit words padded to 32 bits
       Therefore we compress 4 16 bit words = 2 padded dpu words
       down to 3 by 16 bit words = 2 packed 24 bit words i.e.-

       -----------------
       |   0   |   1   |  2 DPU PADDED Words ...
       -----------------
       | 0 | 1 | 2 | 3 |  occupy 4 16 BIT words ...
       -----------------
       |0|1|2|3|4|5|6|7|  or 8 bytes ....
       -----------------
       |1|2|3|5|6|7|      which occupy 6 bytes after stripping ...
       -------------
       | 0 | 1 | 2 |      i.e. 3 16-words ....
       -------------
       |  0  |  1  |      resulting in 2 packed DPU words
       -------------

       Copy resulting 2 packed DPU words into the packet ( = 3*16 words)
       and modify words copied counter accordingly.

       Reset the words accumulated counter

       increment the DPU address counter of the data that has been copied

     If we have accumulated only 2 words

        increment the DPU address counter of the data that has been copied

  Otherwise, if the data is not to be packed (i.e. 16 bit words)

     If we have accumulated 2 by 16 bit words since the last copy
     into the packet operation.

       16 bit data is still padded to 32 bits
       so we extract least significant 16 bit word of the 32 bits
       and copy it into the packet.

       and modify words copied counter accordingly.

       Increment the DPU address corresponding to the DPU data
       about to be copied

       Reset the words accumulated  counter

   If the packet is now full (note that the maximum number of words
   that will be copied must be a multiple of 3 because of the nature
   of the 'packing' operation).

     Output it via routine OUTPUT_DPUMEM.

   Inc pointer within DPU block

procedure OUTPUT_DPUMEM is


  If there are only 2 words in the accumulation buffer
  we are midway thru a packing operation

     So pack what we have

     Copy resulting 1 packed DPU words into the packet ( = 1.5 *16 words padded to 2)
     and modify words copied counter accordingly.

     Reset the words accumulated counter

  Calculate and load the packet length.

  Load Memory Identifier (MID) into Packet Header

  If packet is not empty of RAM data, send it to the telemetry queue.

  Reset words copied counter.
```

```
procedure FLUSH is

   Call OUTPUT_DPUMEM to force output of packet to telemetry queue.

   Reset words copied counter.
```

#### *6.4.2.37 memloc.ads*

Extracted from file "memloc.ads"

```
    Function
    ========

    This file contains the specification only package MEMLOC.
    This package defines any fixed memory locations.

-----------------
package MEMLOC is
-----------------


       Define the location of the ADASCOPE version ID we are running

       Define the size of the telecommand and telemetry queues

       Define RBI Communication Area

       Define the location TC_LOC of the telecommand queue area

       Define the location TM_LOC of the telemetry queue area

       Define other tc/tm special addresses (e.g.. queue pointers)

       Define the location of the filter wheel parameters table

       define BCP4 processing addresses (these are fixed to assist assembler
       and ADA routines to communicate with each other).

       define RBI special addresses

       Define Time Control Flag locations.

       Define the Bootstrap Parameter Area

       define SSI processing addresses.
```

#### *6.4.2.38 modeman.ads*

Extracted from file "modeman.ads"

```
    Function
    ========

    This file contains the specification for the mode manager package.
    This implements mode changes and supplies HK status information.


    Reference
    =========

    Dependencies
    ============

with TYPES; use TYPES;


------------------
package MODEMAN is
------------------


  function TO_MODE(MODE : UINT16; SRC_AND_SEQUENCE_COUNT : UINT16)
                   return BOOLEAN;


      This function implements the mode change mechanism from the
      current mode to the new MODE.

      where :

      MODE is the new mode requested, in the range 0 .. 5
      SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
      of the associated telecommand.

      Returns TRUE if the command was successfully accepted


  function MODE return UINT16;


      This function returns the current mode of the ICU.
```

#### *6.4.2.39 modeman.adb*

Extracted from file "modeman.adb"

```
   ALLOWED TRANSITIONS

   To       | SAFE  | IDLE  | SCI  | ENG  | INT SAFE |
   --------|       |       |      |      |          |
   From     |       |       |      |      |          |
   ----------------------------------------------
   SAFE     | yes   | yes   | no   | no   |    no    |
   IDLE     | yes   | yes   | yes  | yes  |   yes    |
   SCIENCE  | yes   | yes   | yes  | no   |    no    |
   ENG      | yes   | yes   | no   | yes  |    no    |
   INT SAFE| yes   | yes   | no   | no   |    yes   |
   ----------------------------------------------

   The following is the specification of a task internal to this package.


  task SAFING_TASK is

      pragma priority(IMPORTANCE.SAFING);

      entry START(MODE : UINT16; LEVEL : UINT16; SRC_AND_SEQUENCE_COUNT : UINT16);

   end SAFING_TASK;


    where START starts the sequence of commands necesary switch to mode MODE at safe level LEVEL
    and SRC_AND_SEQUENCE_COUNT is the source and sequence count of the requesting telecommand.
    LEVEL can take values DETECTOR.FULL or DETECTOR.HALF_SAFE.

    N.B. The parameters MODE and LEVEL are separate even though MODE implies LEVEL, because
         in earlier releases of the telecommand specification, LEVEL was a sub parameter of
         MODE.

   task body SAFING_TASK is


       Commence infinite loop

          Await a call to the entry point START

             Upon such a call

                Take a copy of the parameters for local use.

            If we are going to full safe

                Disable all SSI output except H/B

                Abort current DPU exposure

             Request HV safing using DETECTOR.PERFORM_HV_SAFING.

             If HK safing proceded OK

                Request F/W Safing using MECHANISM.PERFORM_FW_SAFING

             If we are going to full safe

                Re-enable SSI

                Init DPU

             If all still OK

                Set ICU mode to requested mode by storing it in CURRENT_MODE

  function TO_MODE(MODE : UINT16; SRC_AND_SEQUENCE_COUNT : UINT16)
    return BOOLEAN is


      If mode parameter illegal or not in allowed table then

        If mode out of range then

            Construct illegal mode error packet

         Else if illegal transition then
```

```
         Construct illegal parameter values error packet
      Send unsuccessful acceptance packet and return false
   If next mode is a safe mode then
      Determine whether it is intermmediate or safe
         Initiate the safing sequence using SAFING_TASK.START
         But if the task is already in use
         Send unsuccessful execution packet indicating 'busy' to ground
         Return FALSE
   Otherwise
      If we are switching to Idle but the f/w is not at blocked
         Send 'F/W not at blocked' execution failure  message
         and return with FALSE
      Record mode in CURRENT_MODE and return true
function MODE return UINT16 is

   Return the CURRENT_MODE value
```

***6.4.2.40 mutex.ads***

```
Extracted from file "mutex.ads"


    Function
    ========

    This file contains the specification for the MUTEX package. This provides
    a mutual exclusion semaphore emulation;

----------------
package MUTEX is
----------------


 task type SEMAPHORE is


    entry SEIZE;


        This entry point acquires the resource

    entry RELEASE;


        This entry point releases the resource
```

#### *6.4.2.41 mutex.adb*

```
Extracted from file "mutex.adb"


 Function
 ========

 This file contains the body for the MUTEX package. This provides a
 mutual exclusion semaphore emulation;

----------------------
package body MUTEX is
----------------------


   task body SEMAPHORE is


      Assume, by default, the resource is not in use.

      Begin infinite loop.

        Await a call to seize or release a resource.

          If resource is flagged as not 'in use'

            Allow acceptance of a seize resource request

           accept SEIZE do


              and set flag as 'in use'

          If resource is flagged as 'in use'

            Allow acceptance of a release resource request

           accept RELEASE do


              and set flag as not 'in use'
```

#### *6.4.2.42 nhk.ads*

Extracted from file "nhk.ads"

```
    Function
    ========

    This file contains the specification for package NHK.

    The function of this package is to provide routine(s) to construct and
    place Non-Periodic Housekeeping (NHK) packets into the telemetry queue
    prior to their being transmitted to the ground.


    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

--------------
package NHK is
--------------


  procedure PUT(SUB_TYPE : PACKET.TELEMETRY_SUBTYPE;
                SID_EX   : PACKET.SID_TYPE;
                PARAMS   : UINT16_ARRAY;
                SIZE     : INTEGER);


    The procedure PUT constructs and places an NHK packet in the telemetry
    queue. The interface is as follows:

    where:

    SUB_TYPE specifies the sub-type of NHK packet to be placed in the queue.
             It will take one of the the following values:

             PACKET.EVENT_REPORT          := 1;
             PACKET.EXCEPTION_REPORT      := 2;
             PACKET.MAJOR_ANOMALY_REPORT  := 3;

    SID_EX   specifies the Structure Identifier (SID) to be loaded into the
             packet

    PARAMS   specifies an array of parameters to be loaded into the packet.
             NOTE - the index range of the parameter array should start at 0.

    SIZE     specifies the number of parameters to be loaded from PARAMS.
```

***6.4.2.43 nhk.adb***

Extracted from file "nhk.adb"

```
 Function
 ========

 This package body implements the body for package NHK.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------
package body NHK is
-------------------


   procedure PUT(SUB_TYPE : PACKET.TELEMETRY_SUBTYPE;
                 SID_EX   : PACKET.SID_TYPE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER) is


       Create an instance of the Packet Data Structure.

      If this packet's SID is enabled (use TM_MAN.SID_STATUS)

        Place current time (obtained from TIME_MAN.TIME_STAMP) in data field header

        Flag presence or absence of CRC in data field header

        Calculate and load packet length

        Load in the Structure Identifier (SID)

        Load Number of Parameters

        Load parameters into packet

        Put packet record into queue using TMQ.PUT
```

#### 6.4.2.44 packet.ads

Extracted from file "packet.ads"

```
Function
========

This file contains the specification only package PACKET. This
defines the format of the telecommand and telemetry packets used by the OM
instrument and are derived from the description in the 'Telecommand
and Telemetry Specification', XMM-OM/MSSL/ML/0010.
```

#### *6.4.2.45 peek_poke.ads*

Extracted from file "peek_poke.ads"

```
    Function
    ========

    This file contains the specification for the XMM-OM low-level memory read/write.
    The program is written in assembler and linked as foreign.

--------------------
package PEEK_POKE is
--------------------


  function PEEK(addr: UINT16; addr_state: UINT16) return UINT16;


    This function returns the word stored at address addr in
    address state addr_state

  function POKE(poke_val: UINT16; addr: UINT16; addr_state: UINT16) return UINT16;


    This function puts into memory the word poke_val at the location addr in
    address state addr_state. It returns the word that was poked.
```

#### *6.4.2.46 peek_poke.asm*

```
File is peek_poke.asm

    Name
     peek

    Description
     Picks up an address to be peeked and the Address State from the stack,
     switches to that Address State, peeks the address, selects the
     original Address State and exits with the value peeked in r2.

    Calling sequence
     var := peek(address,address_state);

     (All parameters & return type are UINT16)

    Input
     r0   Link register
     r2   Uplevel register (not needed ?)
     r14  Frame pointer (not needed ?)
     r15  Stack pointer

    Output
     r2   Holds contents of address peeked

    Altered
     r1, r2, r3, r4

    Register map
     r0   Link register
     r1   Holds entry Address State
     r2   Return value
     r3   Holds address to peek
     r4   Holds Address State to switch to

    Notes
     Assembled for use as a foreign code segment in Ada.
     Registers r0-r4 can be trashed.
     All other registers must be preserved.

    Assumptions

     No error checking is performed.

peekaddr

         Save the current address state and change address state
         Read the memory location
         Restore old address state
         Return

    Name
     poke

    Description
     Picks up an address to be poked, the Address State and the value
     to be poked into memory from the stack, switches to that Address
     State, pokes the address, selects the original Address State and
     exits with the value poked in r2.

    Calling sequence
     var := poke(value,address,address_state);

     (All parameters & return type are UINT16)

    Input
     r0   Link register
     r2   Uplevel register (not needed ?)
     r14  Frame pointer (not needed ?)
     r15  Stack pointer

    Output
     r2   Holds value poked into memory

    Altered
     r1, r2, r3, r4

    Register map
     r0   Link register
     r1   Holds entry Address State
     r2   Holds value to poke and return value
```

```
 r3   Holds address to poke
 r4   Holds Address State to switch to

Notes
 Assembled for use as a foreign code segment in Ada.
 Registers r0-r4 can be trashed.
 All other registers must be preserved.
 Is a function because procedure definition in Ada appears
 not to link properly (doesn't see assembler label).

Assumptions
```

No error checking is performed.

pokeaddr

```
Save current address state

Write address with value

Change back to original address state

Return
```

#### *6.4.2.47 rbi.ads*

```
Extracted from file "rbi.ads"


    Function
    ========

    This file contains the specification for the RBI package. This, in turn,
    contains RBI service routines. The package RBI and RBI_INT together
    control and monitor the RBI (Remote Bus Interface).

    The code in this package is based on the description of the RBI chip
    given in "Standard RBI Chip For OBDH Interface
    (MC1031 Technical Informations 2.8-01/06/95 and from the
    "OBDH Bus Protocol Requirements Specification", XM-IF-DOR-0002.

--------------
package RBI is
--------------


  procedure INIT;


      Performs RBI package initialisation.

  function UNCORRECTED_OBT return OBT_TYPE;


      Returns the uncorrected OBT (On-board Time) from the RBI.

  function CORRECT_OBT(UNCORRECTED_OBT_VALUE : in OBT_TYPE) return OBT_TYPE;


      Applies the required correction to the OBT documented in the
      ADV technical note 2.8-01/06/95

  function CORRECTED_OBT return OBT_TYPE;


      Combines the functions of UNCORRECTED_OBT and CORRECT_OBT;

  procedure SET_OBT(OBT_VALUE : in OBT_TYPE);


      Sets the RBI OBT value. This is usually extracted from an Add Time Code
      packet TM(10,3).

  function "+"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE;


      Adds OBTs together N.B. only accurate to 2**-8 secs!!!!
      Now redundant as never used.

  function "-"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE;


      Subtract OBTs N.B. only accurate to 2**-8 secs!!!!
      Now redundant as never used.

  procedure SET_SYNC_READY(SYNC_ENABLE : BOOLEAN);


      Set/Unset Synchronisation Enable Bit in RBI Configuration Register

  task WATCHDOG is
    pragma priority(IMPORTANCE.CPU_RESET);

    entry PARAMS(TIMOUT : UINT16 ;
                 RESET_INTERVAL : UINT16 ;
                 OK : in out BOOLEAN);
    entry ENABLE;
    entry DISABLE;

  end WATCHDOG;


      This task controls the RBI watchdog.

      ENABLE  starts the task.
      DISABLE stops the task.
```

```
    PARAMS resets the time intervals used to control the watchdog.
       TIMOUT specifies what value should be loaded into the
              watchdog timer counter.
       RESET_INTERVAL specifies how often the software should
              reload the time counter with TIMOUT.

function TM_READY return BOOLEAN;


    Returns whether TM_READY (telelemetry ready to transmit)  bit is set
    in the RBI status register

procedure SET_TM_READY(SET_TO_ON : BOOLEAN);


    Set/Unset TM_READY (telelemetry ready to transmit) bit in the
    RBI status register

procedure TOGGLE_TM_READY;


    Toggles TM_READY (telelemetry ready to transmit) bit in the
    RBI status register

function TC_READY return BOOLEAN;


    Returns whether TC_READY (ready to receive telecommand) bit is set
    in the RBI status register

procedure SET_TC_READY(SET_TO_ON : BOOLEAN);


    Set/Unset TC_READY (ready to receive telecommand) bit in status register

procedure SET_COMM_AREA_TM_INFO(START_ADDRESS : UINT16;
                                PACKET_LENGTH : UINT16);


    Store start address and length of a telemetry packet in
    the communications area (CCA).

procedure SET_COMM_AREA_TC_INFO(START_ADDRESS : UINT16);


    Store start address of where the telecommmand should be stored
    in the communication area (CCA).

function STATUS_REGISTER return UINT16;


    Returns the RBI Status Register

function CONFIG_REGISTER return UINT16;


    Returns the RBI Configuration register
```

#### *6.4.2.48 rbi.adb*

Extracted from file "rbi.adb"

```
 Function
 ========

 This file contains the body for the RBI package. This, in turn,
 contains RBI service routines. The package RBI and RBI_INT together
 control and monitor the RBI (Remote Bus Interface).

 The code in this package is based on the description of the RBI chip
 given in "Standard RBI Chip For OBDH Interface
 (MC1031 Technical Informations 2.8-01/06/95 and from the
 "OBDH Bus Protocol Requirements Specification", XM-IF-DOR-0002.

-------------------
package body RBI is
-------------------

   Contents of RBI OBT (On-Board Time) as follows:

        -------------------------------
        | OBT 0   |  OBT 1  |  OBT 2  |  OBT location
        -------------------------------
        |    C    |    D    |    E    |  Register
        -------------------------------
        |0      15|16     31|32-42|xxx|  Bits in Counter
        -------------------------------
        |    SECS       |    FRAC     |  Secs/Fractions of sec
        -------------------------------
        |23            0|-1     -19|xxx|  2**? secs
        -------------------------------

    Note the layout of the SCET (Spacecraft Elapsed Time)
    in a packet for comparison (and its offset).

         23            0 -1     -16
   -------------------------------
   |    Coarse Time     |  Fine   |
   -------------------------------

   Create a semaphore to control access to the freeze register by creating an instance
   of the SEMAPHORE task in package MUTEX called FREEZE_REGISTER.

  function TO_OBT_TYPE(INPUT : in LONG_INTEGER) return OBT_TYPE;
  function TO_LONG_INT(INPUT : in OBT_TYPE)     return LONG_INTEGER;


   The above internal routines are used to convert an OBT to or from LONG_INTEGER

  function UNCORRECTED_OBT return OBT_TYPE is


      Ensure exclusive use of RBI configuration register
      while we peform a Freeze operation using the SEIZE entry in MUTEX.

      "Freeze" the current time by writing appropriate instruction
      to the RBI configuration register.

      Release the register for use by other code by using RELEASE entry in MUTEX.

      Read and store bits 0-15 of the result.

      Read and store bits 16-31 of the result

      Read and store remaining bits 32-42 (result in high order bits)

      Return the stored result (i.e. the OBT as defined above).

  function CORRECT_OBT(UNCORRECTED_OBT_VALUE : in OBT_TYPE) return OBT_TYPE is


      If bits 32 to 42 of the uncorrected OBT is greater than 3ff hex

         subtract 1 from bits 0 to 31

      return the result (a corrected OBT).

  function CORRECTED_OBT return OBT_TYPE is
```

```
   Get the OBT and correct it using CORRECT_OBT.

procedure SET_OBT(OBT_VALUE : in OBT_TYPE) is


   Prevent use of Freeze register by other code
  while we do this using FREEZE_REGISTER.SEIZE

   Write the most significant 16 bits of the provided OBT
   into the 1st RBI OBT update register

   Write the next 16 bits of the provided OBT
   into the 2nd RBI OBT update register

   Release Freeze register using FREEZE_REGISTER.RELEASE.

function "+"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE is


    Prevent Overflows on additions.

    Add the two supplied OBT's after conversion using TO_LONG_INT
    and return the result as an OBT using TO_OBT_TYPE .

function "-"(A : OBT_TYPE; B : OBT_TYPE) return OBT_TYPE is


    Prevent Overflows on subtractions.

    Subtract the two supplied OBT's after conversion using TO_LONG_INT
    and return the result as an OBT using TO_OBT_TYPE .

function TO_OBT_TYPE(INPUT : in LONG_INTEGER) return OBT_TYPE is


    This routine is used internal to the package to convert
    a supplied 48 bit integer (stored in a signed 64 bit integer)
    into an OBT format (3*16 bit words).
    The value is only accurate to 2**-8 seconds.

   Split up the 64 bit word into 3 * 16 words using appropriate bit shifing and masking

   The MSW contains the 16 high order bits of the least significant 32 bits
   The next word contains the least significant 16 bits
   The last word is set to zero as it represents value < 2**-8 seconds

function TO_LONG_INT(INPUT : in OBT_TYPE) return LONG_INTEGER is


    This routine is used internally to the package to convert
    a supplied OBT (3*16 bit words) into a 64 bit integer.

   Ignore the least signifcant word as it represents values < 2**-8 seconds.
   Concatenate the Most Signifcant word and the next to form a 32 bit value.
   Return the result as a 64 bits signed integer.

procedure SET_SYNC_READY(SYNC_ENABLE : BOOLEAN) is


   Get the RBI configuration register value

   If its Synchronisation Enable bit is not as requested by SYNC_ENABLE

     toggle it

task body WATCHDOG is


   Begin infinite loop

       Await a call to one of the rendevous points

         If a call to the set params entry point PARAMS is made

            If the parameters are inconsistent or invalid

               Flag as invalid and don't store.

            Otherwise

               Store the specified timout period (units = 1/256 secs )
```

```
             and reset interval (units = secs) supplied.

             Flag as valid.

    Or

        If a call to enable the watchdog is made via entry ENABLE

          Determine if watchdog is already enabled
          from the RBI configuration register

          Write the stored timout period to appropriate register

          If the watchdog is not already enabled,  enable the watchog

            by toggling the appropriate bit in the confuration register.

      Or

        If a call to disable the watchdog is made via entry DISABLE

          Determine if watchdog is enabled by examining the RBI configuration register

          If it's no aleady disabled, disable it

            by toggling the appropriate bit in the confuration register.

      Or

        Provided the watchdog is enabled

          and if no call to a rendevous is made for the stored reset period

          Reset counter in watchdog (thus as long as the ICU code
          is running, the timout counter is never allowed to get
          to zero) by writing to the appropriate RBI register.

procedure INIT is


    Set up the communication area by writing its address shifted to the right by 7
    to the RBI Base Address Register.

    Ensure TC and TM ready flags are disabled for now
    using SET_TC_READT and SET_TM_READY.

function TM_READY return BOOLEAN is


   Get the RBI Status register value

   Extract and return the status of the TM_READY bit

procedure SET_TM_READY(SET_TO_ON : BOOLEAN) is


    Use TM_READY to see if
    the telemetry ready for transmission bit is not
    already in the status requested by SET_TO_ON.

       If it isn't
       oggle it so it is using TOGGLE_TM_READY.

procedure TOGGLE_TM_READY is


    Toggle the current RBI TM_READY (telemetry ready for transmission)
    flag state by writing the appropriate bit to the RBI configuration register.

function TC_READY return BOOLEAN is


   Get RBI status register value

   Extract and return the TC_READY
   (ready to receive a telecommand) bit status.

procedure SET_TC_READY(SET_TO_ON : BOOLEAN) is


   Get current status RBI register.

   If its bit 11 (the TC_READY- ready to receive a telecommand) is
```

```
   already in the status requested by SET_TO_ON

     Do nothing

     Otherwise if it needs to be on

       Set it on within the RBI status word read back earlier

     else

       Clear it within RBI status read back earlier.

     Finally, write back the resulting RBI status word to the
     register (NOTE: only bits 11-15 are written to)

procedure SET_COMM_AREA_TM_INFO(START_ADDRESS : UINT16;
                                PACKET_LENGTH : UINT16) is


    Store the start address START_ADDRESS of the TM packet in bytes,
    relative to the start adddress of the CCA, in the CCA,

   Store the packet length PACKET_LENGTH in the CCA in words but
   with 1 subtracted and the MSB set, as per specification.

procedure SET_COMM_AREA_TC_INFO(START_ADDRESS : UINT16) is


    Store the TC packet start address START_ADDRESS in bytes relative to the start
    of the CCA, in the CCA.

function CONFIG_REGISTER return UINT16 is


   Get the RBI configuration register value.

function STATUS_REGISTER return UINT16 is


   Get the RBI status register value.
```

#### *6.4.2.49 rbi_ih.ads*

Extracted from file "rbi_ih.ads"

```
Function
========

This file contains the specification for the XMM-OM rbi interrupt handler.
The interrupt handler is written in assembler and linked as foreign.
```

#### *6.4.2.50 rbi_ih.asm*

File is rbi_ih.asm

```
        This follows closely the document:
        OBDH Bus Protocol Requirement Specification
        XM-IF-DOR-0002
          Fetch the interrupt counter
          Check for impending overflow
          If it's OK, increment it
          otherwise avoid overflow
          read config_reg
          get the bits we're interested in
          is it lossn (0)?
          is it instruction to user (1)?
          is it instruction to rbi (2)?
          is it other_it (3)?


          otherwise serious error so safe


          Read value from appropriate register
          (which also clears the interrupt)
          read instruction to user reg
          If the register is 0, jump to tcq_add
          when it's an Instruction to RBI interrupt


          read instruction to rbi reg
          This could be caused by warm reset and we
          call back into the bootstrap (TBI)


          If it's any other sort of interrupt
          This is an error (so we safe or discard with exception, TBD)
          and finish off
         -------------------------------------------------------------------
          tcq_add ***********************************************************
         -------------------------------------------------------------------
          set tc_ready to false
          if full
            Tell s/c we can't accept packets (This ought never happen as we take packets away in
time?)
            read input_pointer from memory
            add one
            mod it with no_tc_slots
            keep for future
            store it again
            Now set up new address for next packet
            start_address = 16#404# + r0*248
          if not tc_q.is_full
          i.e.
          if (input_pointer+1)&3 != output_pointer
            (increment input_pointer)
            the required mask is 0
          else required mask = set_tc_ready_mask (16#0010#)
            Read status
            'and' this status with set_tc_ready_mask (16#0010#);
            Compare this with the required mask
            If they're the same, finish off
            if REQUIRED_MASK = SET_TC_READY_MASK (16#0010#)
              'or' the status that was read with set_tc_ready_mask (16#0010#)
            else 'and' the status that was read with clear_tc_ready_mask (16#ffef#)
              xio this to the rbi_status reg
              finish off
          Read status
          If the tm_ready bit is set
            write a reset output transfer request to the rbi config reg
          Increment the output_pointer
          Read the input_pointer and compare output_pointer with input_pointer
          If they're equal
            finish off
          Otherwise calculate the address and write it to cca_tm_start
          Calculate the length and write it to cca_tm_length
          Read the RBI status
          'and' it with the tm_ready_mask (16#0080#)
          finish off
          if zero, write a reset_output_transfer_request to the RBI config reg
          finish off
          Tidy up after finishing
          FINISH OFF:
```

```
Recover registers
Turn on interrupts
Back from whence we came
```

#### ***6.4.2.51 reset.ads***

Extracted from file "reset.ads"

```
    Function
    ========

    This file contains the specifications for the XMM-OM reset package.
    reset itself is written in assembler and linked as a foreign code function.

    Reference
    =========

    Dependencies
    ============

with TYPES; use TYPES;


----------------
package RESET is
----------------


  function reset(addr : UINT16) return UINT16;


      This function jumps to the address given on its argument list

      where :

      addr is the address of a routine to jump to
```

#### *6.4.2.52 reset.asm*

```
File is reset.asm

    Name
     reset

    Description
     When called, enables the start up ROM and jumps to
     location zero.

    Calling sequence
     var  :    UINT16
     addr :    UINT16;

     var := reset(addr);

    Input
     r0 Link register
     r2 Uplevel register (not needed ?)
     r14       Frame pointer (not needed ?)
     r15       Stack pointer

    Output
     Does not return

    Altered
     Everything

    Register map
     r0, r1, r2                                                  Working register
     r3        Holds parameter to routine

    Notes
     Assembled for use as a foreign code segment in Ada.
     If addr = 0 then the start up rom is enabled and a jump
     to 0 is performed.
     Any other value for addr and the start up rom is left as it is
     and the jump to the address specified is made. 6 words (the
     floating pt overflow, fixed pt overflow and timer b interrupt
     vectors are copied from a buffer starting at 16#03FA# to their
     proper locations (16#0026#,16#0028# and 16#0032# respectively)
     before the jump.
     Interrupts are disabled during this routine and page 0 is
     mapped in.

    Assumptions

     No error checking is performed.


resetentry

     Disable all interrupts
     Stop timer B
     Make sure we are in address state 0
     Get parameter from stack
     If parameter is equal to zero ...
     ... then branch to RESTART
     Copy new interrupt vectors to data space
     Copy new interrupt vectors to page 3
     Reselect page 0
     Clear all interrupts and machine errors
     Now start op code

RESTART

     Jump to warm reset code
```

#### *6.4.2.53 science_fm.ads*

Extracted from file "science_fm.ads"

```
    Function
    ========

    This file contains the specification for the SCIENCE_FM package.

    The function of this package is to provide routine(s) to construct and
    place Science packets into the telemetry queue prior to their being
    transmitted to the ground.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010, Section 3.9.

----------------------
package SCIENCE_FM is
----------------------


  procedure PRIORITY_DATA(SID_EX          : PACKET.SID_TYPE;
                          DPU_DATA        : UINT16_ARRAY);


    This routines constructs and places science packets in the telemetry
    queue derived from the supplied DPU Priority Data.

    where:

    SID_EX   specifies the Structure Identifier (SID) to be loaded into the
             packet

    DPU_DATA contains the DPU Priority Data record for loading into the packet.
             Note 1) the index range of DPU_DATA should start at 0.
                  2) the length of data to be loaded in the packet is implied
                     by the contents of DPU_DATA(1). This states the number
                     of following words that are to be included i.e. it
                     conforms to the usual DPU data record conventions.

  procedure AUXILIARY_DATA(SID_EX          : PACKET.SID_TYPE;
                           DPU_DATA        : UINT16_ARRAY);


    This routines constructs and places science packets in the telemetry
    queue derived from the supplied DPU Auxiliary Data.

    where:

    SID_EX   specifies the Structure Identifier (SID) to be loaded into the
             packet

    DPU_DATA contains the DPU Auxiliary Data for loading into the packet.
             Note 1) the index range of DPU_DATA should start at 0.
                  2) the length of data to be loaded in the packet is implied
                     by the contents of DPU_DATA(1). This states the number
                     of following words that are to be included i.e. it
                     conforms to the usual DPU data record conventions.

  procedure REGULAR_DATA(SID_EX          : PACKET.SID_TYPE;
                         DPU_DATA        : UINT16_ARRAY);


    This routines constructs and places science packets in the telemetry
    queue derived from the supplied DPU Regular Data.

    where:

    SID_EX   specifies the Structure Identifier (SID) to be loaded into the
             packet

    DPU_DATA contains the DPU Regular Data record to be loaded into the packet.
             Note 1) the index range of DPU_DATA should start at 0.
                  2) the length of data to be loaded in the packet is implied
                     by the contents of DPU_DATA(1). This states the number
                     of following words that are to be included i.e. it
                     conforms to the usual DPU data record conventions.
```

```
procedure FLUSH(SID_EX : PACKET.SID_TYPE);

    Flushes Regular Science Data Output Buffer upon receipt of the
    'end of data' alert from the DPU. This is required because Regular
    Science Data is spread across many DPU blocks (i.e. it is not
    confined to one DPU block).
```

#### *6.4.2.54 science_fm.adb*

```
Extracted from file "science_fm.adb"


 Function
 ========

 This package body implements the specification given in SCIENCE_FM.ADS

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010, Section 3.9.

---------------------------
package body SCIENCE_FM is
--------------------------


    Create Instances of Priority, Auxiliary and Regular Science
    Packet Records to be used as working storage.

    Specify internal routines used by package.

   procedure OUTPUT_SCIENCE(TERMINATE_GROUP : BOOLEAN;
                            SCIENCE_PACKET  : in out PACKET.TM_TYPE;
                            PACKET_NUMBER   : in out UINT16;
                            DPU_BLOCK       : in out UINT16;
                            WORDS_COPIED    : in out INTEGER;
                            SID_EX          : PACKET.SID_TYPE);


    This procedure constructs the packet header and sends the resulting packet
    to the telemetry queue via TMQ.PUT.

    TERMINATE_GROUP if true indicates it will be the last packet of a group
    SCIENCE_PACKET is the packet to be sent.
    PACKET_NUMBER is the number of the packet within a group.
    DPU_BLOCK is the number of DPU blocks within a group.
    WORDS_COPIED is the number of DPU data words copied into the current packet.
    SID_EX is the Structure Identifier to be placed in the Packet.

   procedure PRI_REG_DATA(PRI_REG_PACKET : in out PACKET.TM_TYPE;
                          PACKET_NUMBER  : in out UINT16;
                          DPU_BLOCK      : in out UINT16;
                          WORDS_COPIED   : in out INTEGER;
                          SID_EX         : PACKET.SID_TYPE;
                          DPU_DATA       : UINT16_ARRAY);


    This procedure constructs science data packets from the DPU data blocks
    sends the resulting packets to OUTPUT_SCIENCE.

    PRI_REG_PACKET is the packet being constructed.
    PACKET_NUMBER is the number of the packet within a group.
    DPU_BLOCK is the number of DPU blocks within a group.
    WORDS_COPIED is the number of DPU data words copied into the current packet.
    SID_EX is the Structure Identifier to be placed in the Packet.
    DPU_DATA is the input DPU block.

   procedure REGULAR_DATA(SID_EX          : PACKET.SID_TYPE;
                          DPU_DATA        : UINT16_ARRAY) is


      Simply pass the DPU block to the PRI_REG_DATA routine.

   procedure PRIORITY_DATA(SID_EX          : PACKET.SID_TYPE;
                           DPU_DATA        : UINT16_ARRAY) is


      Simply pass the DPU block to the PRI_REG_DATA routine.

   procedure PRI_REG_DATA(PRI_REG_PACKET : in out PACKET.TM_TYPE;
                          PACKET_NUMBER  : in out UINT16;
                          DPU_BLOCK      : in out UINT16;
                          WORDS_COPIED   : in out INTEGER;
                          SID_EX         : PACKET.SID_TYPE;
                          DPU_DATA       : UINT16_ARRAY) is
```

```
    Determine whether this is regular or priority data.

    DPU priority data blocks are split up and output
    across several packets. The resulting collection of packets is
    a complete 'group' of packets.

    DPU regular data blocks are split up and output
    across several packets. The resulting collection of packets is
    only part of a 'group' of packets. The FLUSH command terminates
    the group.

    Calc number of words to copy from DPU block into current packet

    If this is the second DPU block to be copied into
    the current group  of packets

       Set up the offset within the science sub-header accordingly.

    Count how may DPU blocks we have processed for this group so far.

    Loop over data to be copied

      Copy a word into the current packet

      If the current packet is now full

        Flag that we should terminate the group if this is priority
        science data and all data has been copied into the packet.

        Give the packet to the science output routine with the
        terminate group flag appropriately set by calling OUTPUT_SCIENCE.

    If it is priority science data

      Give the packet to the OUTPUT_SCIENCE routine with
      the 'terminate group' flag set.

    If it's regular data

      Give the packet to the OUTPUT_SCIENCE routine with
      the 'terminate group' flag *NOT* set.

procedure AUXILIARY_DATA(SID_EX          : PACKET.SID_TYPE;
                         DPU_DATA        : UINT16_ARRAY) is


    DPU auxiliary data blocks are buffered up into 1 packet.
    The result is a standalone packet.

    Calc number of words to copy from DPU block into the packet.

    If this block will exceed the current packet capacity

      Give the packet to the OUTPUT_SCIENCE routine with
      the 'terminate group' flag set to true

    If this is the 1st block to be copied into the packet

       Set up a dummy offset in science sub-header (offset = FF (hex))

    Count the number of auxiliary blocks processed so far.

    Loop over data to be copied

       Copy (and count) a word into the current packet

procedure OUTPUT_SCIENCE(TERMINATE_GROUP : BOOLEAN;
                         SCIENCE_PACKET  : in out PACKET.TM_TYPE;
                         PACKET_NUMBER   : in out UINT16;
                         DPU_BLOCK       : in out UINT16;
                         WORDS_COPIED    : in out INTEGER;
                         SID_EX          : PACKET.SID_TYPE) is


    Build the header

    Calculate and load the packet length.

    If 'terminate the group' flag is set

       and it's the first packet so far for the group

          flag it as standalone
```

```
        otherwise set segmentation flag to indicate it is the last
        packet of a group.

        and also ensure science sub header offset is all 1's

  otherwise, if we are not terminating the group

     and is the first packet

        Set segmentation flag to indicate first packet of group

     otherwise

        Set segmentation flag to indicate continuation packet of group

  Load Structure Identifier (SID) into Packet Header

  Load group count with the packet number count within the group.

  In the special case of regular data

      Change the sub-type as per SID

  Send out packet to the telemetry queue using TMQ.PUT

  Modify counters etc according to whether we are
  terminating the group

procedure FLUSH(SID_EX         : PACKET.SID_TYPE) is


  Call OUTPUT_SCIENCE routine with 'terminate group' set to true
```

#### *6.4.2.55 ssi_driver.ads*

Extracted from file "ssi_driver.ads"

```
   ------------------------
   procedure SSI_INTERRUPT;
   ------------------------


       SSI_INTERRUPT is the SSI interrupt handler (written in Ada but
       connected via the assembly code ssi_ih.asm)

   ---------------------------------------------------
   procedure GET(DATUM : out UINT16; RET : out INT16);
   ---------------------------------------------------


       This gets one word of data from the SSI (from the DPU)
       DATUM is the word
       RET is a signed 16-bit word which is
       0 if there are no words to read
       1 if there is a word to read
       <0 if there was an error

   ----------------
   procedure RESET;
   ----------------


       This procedure resets the SSI link
      (software only---there is no hardware reset)

   ---------------------------------------------------------
   function PUT(BUFFER_DATA : in UINT16_ARRAY) return INT16;
   ---------------------------------------------------------


       This puts an array of words on the SSI (to the DPU)
       BUFFER_DATA is an array of unsigned 16-bit words of data
       returns a signed 16-bit integer which is
       0 if successful
       <0 if there was an error

   SSI_INT_COUNT : UINT16 := 0;


       This variable is a counter for the number of SSI interrupts received
       It wraps back to 0 after 0xffff

   ERROR_COUNT : INT16 := 0;


       This variable is a counter for the number of SSI errors that have occured
       When it reaches 255 it stays at 255
```

#### *6.4.2.56 ssi_driver.adb*

**Extracted from file "ssi_driver.adb"**

```
 Function
 ========

 This file contains the body for package ssi_driver.
 It writes to and reads from the SSI interface.

 Reference
 =========

 The SSI interface is described in http://mssls7.mssl.ucl.ac.uk/sw/ssi.html
 and here ...

 The complete description:



                                    SSI

                        Serial Synchronous Interface

Overview

The SSI is a bi-directional communications interface between the DPU and ICU
which is carried on the DEM backplane.

The definition of the SSI is in XMM-OM/MSSL/SP/0007 "Electrical Interfaces
Specification".

Hardware

Both the ICU and the DPU can send and receive data on this interface but the
ICU is the master.

The interface consists of:

   * SSI_CLK: a continuous clock signal generated by the ICU
   * SSI_ENV_TX: active high when data present
   * SSI_DATA_TX: 16-bit data
   * SSI_ENV_RX: active high when data present
   * SSI_DATA_RX: 16-bit data
   * Signal return

Commands are sent from the ICU to the DPU. Science data is passed from the
DPU to the ICU when demanded by the ICU. Alerts are sent (unrequested) by
the DPU to the ICU. There is no direct feedback as part of the protocol and
there is no error correction nor checksums. The interface can be thought of
as the same irrespective of direction.

The SSI clock frequency is 125 kHz producing a period of 8 us (1
bit-period). The SSI 16-bit data words are separated by at least one
bit-period and at most the SSI block gap (defined in software). The SSI data
blocks are separated by at least the SSI block gap (defined in software).

Transmitting data

The words that constitute the block are sent not more than the SSI block gap
apart and, when finished, the software must wait for at least the SSI block
gap before sending more data. The receiving software must wait for a little
longer than the transmitting software's block gap to be sure to see the gap.
A factor of two is sufficient.

Receiving data

The data being received must be read suitably fast and if the time between
any two words is greater than the SSI block gap, the gap will be considered
a block gap. All blocks contain a length as their second word so errors
caused by an accidentally lengthened word gap may be identified (see data
format).

SSI block gaps

Because the SSI block gaps are defined and used only in software they can be
set to different values in different versions of the code and they can be
different depending on the direction of the data (ICU->DPU or DPU->ICU).

 SSI block gaps as defined by the ICU
              software
```

```
          EPROM code Uploadable code
 ICU -> DPU   >4 ms        >4 ms
 DPU -> ICU   6 ms          4 ms

 SSI block gaps as defined by the DPU
              software

           EPROM code Uploadable code
 ICU -> DPU 2 +/- 1 ms    2 +/- 1 ms
 DPU -> ICU15 +/- 1 ms   15 +/- 1 ms
```

The ICU's SSI hardware will give an interrupt (used by the ICU's software) at the end of the first word of each block. The ICU software must then read this first word before the end of the second word. The time for this is 16 bit-periods for the word and a minimum of 1 bit-period for the word gap. So the software must be able to respond to the interrupt and read the word in 136 us.

An overflow (OVF) bit in the hardware SSI status word is made active (low) if a data word is not read before the arrival of another.

---

SSI errors

If the DPU resets whilst transmitting the first part of a word, that word will be truncated and the envelope will be truncated resulting in an earlier than expected "data receive" flag which will not be able to be processed in time and cause an overflow on the ICU.

If the DPU resets whilst transmitting the last part of a word, that word and the envelope will be truncated but not so much that the ICU's software cannot keep up as in the previous case. This will result in a corrupt last word and, except in the case of a reset during the last word, a truncated SSI block. This will be detected and handled properly by the ICU's software.

---

Data format

The data format is described in XMM-OM ICU-DPU Protocol Definitions Each SSI data block consists of

1. 16-bit type - the block type
2. 16-bit length - the number of 16-bit words following this word (i.e. total length - 2)
3. the rest of the data

The data types are grouped into categories as follows:

Regular DPU to ICU data blocks
: Regular science data.

DPU priority data
: These contain science data that is sent out as soon as it is available rather than at the end of an exposure.

DPU RAM dumps
: RAM dumps.

DPU to ICU alerts
: Alerts from the DPU to signify something is has happened, is ready or an error has occured.

ICU to DPU commands
: Commands to the DPU.

---

Further detail on the ICU software

The first, fast part of the SSI interrupt handler is written in assembler (the first word of the SSI block is read) and the rest in written in Ada (the reading of the rest of the words in the block and the timeout.)

```
   SSI status
    register

 D_TX     2**4
 DATA_FULL2**3
 OVF      2**2
 D_RX     2**1
 INT      2**0
```

Sequence of actions

* SSI INTERRUPT happens
* Read first word (from i/o address f241h) into input software fifo in less than 136 us after the interrupt

```
   * Remember location where next word will be stored for a later check
   * Start stopwatch
   * Set interrupt mask to only allow RBI interrupts
   * Enable interrupts but don't get interrupted for too long!
   * loop
        o read SSI status (i/o address f240h)
        o if the DATA_FULL bit (2**3) is set and there is data to output
             + write a data word to output i/o address (7241h)
        o if input software fifo is full
             + error
        o if D_RX bit is reset
             + read input word (i/o address f241h) into input software fifo
             + re-start stopwatch because there is still data on input
        o else
             + if stopwatch is after 4 ms
                  + break out of loop
        o read ssi status word (i/o address f240h)
        o if OVF bit (2**2) is 0
             + clear overflow (write fffb (hex) to status register i/o
               address 7240h)
             + read a word (from i/o address f241h) and dispose of it
   * end loop
   *
   * read the second word (length) of this SSI block from the software input
     buffer
   * if it is greater then 1027
        o error
   * if no of words read doesn't equal the value of the second word (see
     above) minus 2
        o error
   * read ssi status word (i/o address f240h)
   * if OVF bit (2**2) is 0
        o clear overflow (write fffb (hex) to status register i/o address
          7240h)
        o read a word (from i/o address f241h) and dispose of it
   * clear SSI interrupt by writing fffe (hex) to the SSI status i/o address
     7240h

To Reset

   * reset software input and output fifos and error value
   * write OVR_WR fffb (hex) to status address 7240 (hex)
   * write INT_WR fffe (hex) to status address 7240 (hex)

SSI error codes

error = C
     The SSI input circular buffer has filled so fast or not been emptied
     fast enough and incoming data is about to overwrite outgoing data.
error = 2
     The word count is too large while receiving data in the block. The
     number of words has exceeded that indicated by the second "block
     length" word or has exceeded the maximum allowed (1029).
error = 8
     An overflow (OVF) has been indicated by the ICU's SSI hardware.
error = 7
     An overflow occured at the end of the block.
error = 11
     The second word of the block indicated a length which exceeds the
     maximum allowed (1029).
error = 1
     The length indicated by the second word is inconsistent with the real
     length of the block.
error = 89
     An overflow was found during SSI_DRIVER.PUT
error = 9
     The length found in SSI_DRIVER.PUT exceeded the maximum allowed (1029).
error = b
     The output block length in SSI_DRIVER.PUT exceeded the maximum allowed
     (1029).
```

---

Further detail on the DPU software

The DSP converts a serial SSI word to parallel word. Each received word generates an interrupt. The SSI ISR pushes the word into a circular buffer. The 1ms ISR checks the COLLECTING_A_COMMAND bit. If it is zero (cleared), it decrementes the delay count (stopwatch), else the delaycount (stopwatch) is reset. When the delaycount reaches 0, it is assumed a valid comand has been received (a full block has been received), and the command interpreter is called. The command interpreter checks for integrity of command: it checks the block has:

```
   * a valid command ID
   * a legal length for command ID

It does not count the number of words received and compare this with the
length stored as the second word. The command interpreter is written in C
and the rest of the SSI code in assembler.

On a hardware error the code will:

   * Reset fill pointer.
   * Send out bad block.

  ------------------------------------------------------------------------

 Dependencies
 ============

with SYSTEM;
with UNCHECKED_CONVERSION;
with INTRINSICS;
with ARTCLIENT;

with DEBUG;
with MEMLOC;
with NHK;
with PACKET;


 Suppress all checks to speed up

--------------------------
package body SSI_DRIVER is
--------------------------


    The first word of an SSI block read back by the ssi_ih interrupt handler
    is stored at MEMLOC.SSI_FIRST_WORD_LOCATION for speed.

   procedure SSI_INTERRUPT is


       This (Ada code) is called from ssi_ih.asm (assembler code)

       interrupts are already disabled by the 31750's microcode

       - Read Data -

       Read first word of SSI block from the special address that
       the assembler code (ssi_ih) wrote to

       increment the input buffer pointer

       and wrap it round if necessary

       set the word count for this block to 1

       remember the pointer position for checking the dpu block length later

       remember the initial timer B value

       Turn on RBI interrupts

       loop

          get the SSI status

          if the status shows !data_full and there's some data to send - send it

                and increment the output buffer pointer

          Check to see if the input buffer pointer has wrapped right round
          to the point at which the same buffer should be read from

          If they're too close, store an error "-C" ready for the next time
          something is called

          If there's more data to read - read it

             and increment and wrap round the input buffer pointer

             if the count of words in this block gets far too large, store an error "-2"
```

```
                otherwise increment the READ count

          reset the old stored value of timer B because we haven't
          stopped receiving data yet

          but if there's nothing to read this time round
          check the timer

          if timer B has wrapped round, add on 64K

          exit the loop when we've been waiting to read something for
          26 timer-B ticks (4 ms) i.e. 40 to-spec ticks

       read the SSI status

       if there's been an overflow

          clear the overflow

          do a dummy read to clear

          store an error "-8"

       end loop

    read the SSI status

    if there's been an overflow

       clear the overflow

       do a dummy read to clear

       store an error "-7"

    get the second word of the SSI block from the output buffer
    this contains the number of words minus two that should be in the block

    if the number read is just too large

       remember an error "-11"

       if the length doesn't match the number of words read back
       remember an error "-1"

    clear SSI interrupt by writing to the SSI interface
    as long as the DPU isn't spewing too-long blocks

procedure GET(DATUM : out UINT16; RET : out INT16) is

    returns length

    If there's been an error in the driver part,

       increment the error counter

       and return the error

       Otherwise, read the SSI status

       If there's nothing to read, return 0

       If there's something to read, read it

       incremet the pointer

       and wrap it round

       return the length (1)

function PUT(BUFFER_DATA : in UINT16_ARRAY) return INT16 is

    If there's been an error

       increment the error count

    Read the SSI status
```

#### 6.4.2.57 ssi_ih.ads

Extracted from file "ssi_ih.ads"

```
    Function
    ========

    This file contains the specification for the XMM-OM ssi interrupt handler.
    The interrupt handler is written in assembler and linked as foreign.
```

#### *6.4.2.58 ssi_ih.asm*

File is ssi_ih.asm

```
Sort out the stack
Read first word of SSI block from DPU to ICU and store for Ada
Jump to Ada SSI interrupt handler
  Tidy up
  Return from interrupt
```

#### *6.4.2.59 ssi_in.ads*

Extracted from file "ssi_in.ads"

```
    Function
    ========

    This file contains the specification for package SSI_IN

    The package is used to allow access to the ssi driver code
    in order to receive blocks sent from the DPU.

-----------------
package SSI_IN is
-----------------


  procedure GET(DATA : out UINT16_ARRAY; SUCCESS : out INT16);


      where:

      DATA contains a DPU block sent from the DPU via the SSI interface

      SUCCESS  returns the completion code
               <0 indicates an error.
               >0 indicates success.

  procedure RESET;


      resets the SSI interface.
```

#### *6.4.2.60 ssi_in.adb*

```
Extracted from file "ssi_in.adb"


 Function
 ========

 This file contains the package body for the SSI_IN package

----------------------
package body SSI_IN is
----------------------


  procedure GET(DATA : out UINT16_ARRAY ; SUCCESS : out INT16)  is


      In order to follow the logic of this code, you must be aware that
      the data block received from the DPU via the SSI interface has the
      following format.

      ++++++++++++++++++++++++++++++++++++++++++
      + Word 0 + Word 1 + Word 2  ->  Word N+2 +
      ++++++++++++++++++++++++++++++++++++++++++
      + Block  + Word   +                      +
      +  ID    + Count  +    DPU Data Block    +
      +        +   N    +                      +
      ++++++++++++++++++++++++++++++++++++++++++

      Initialise the word count to 2.

      Initialize the state of the code to be 'at Block ID'

     Commence infinite loop

        Exit from loop when word count is zero (initialised to 2)
        as this indicates we are at end of block.

        Get a datum from the SSI interface, noting
        completion code, using SSI_DRIVER.GET.

        If the completion code indicates a good datum was found
        (i.e. it is greater than zero).

           Now perform action depending on the 'state' of the routine
             (initially at Block ID)

             When the routine is in state 'At Block ID'

                We ought to be at the start of a valid DPU data block
                so check the datum received is a valid DPU header code

                   If it is valid, store the datum in the
                   1st location of the output array

                   and change state of routine to 'at block size'

                Otherwise

                   Prepare and send and SSI exception report.

                   Force end of block condition by setting word count to zero

                   and reset the interface using RESET

             When the routine is in state 'at block size'

                Reset the word count to be the value of the datum.

                Store the datum in the 2nd location in the output array

                Change the routine state to 'in block data'

             When the routine is in state 'in block data'

                Store the datum in successive locations in the output array

                Decrement the word count by one.

       Else, if no data was found in the SSI driver queue
       (i.e. the completion code was zero).
```

```
        Wait a bit

      Else

        exit from the loop as we have an error

      end of infinite loop

    If the completion code indicates an error
    (i.e. is less than zero).

      Store the completion code in SUCCESS.

      Prepare and send appropriate SSI Exception Report NHK packet

      Reset the interface using RESET

    Otherwise

      Set SUCCESS to 1 to indicate all OK.

    Return from routine

procedure RESET is


    Simply perform a direct call to the low level ssi driver
    reset RESET
```

#### *6.4.2.61 ssi_out.ads*

Extracted from file "ssi_out.ads"

```
    Function
    ========

    The file contains the specification for package SSI_OUT. This package
    controls access to the SSI driver for output, allowing only one
    external object to access the driver code, and therefore in turn the SSI
    interface, at any given moment.

    This package will be merged with the SSI_IN package
    in the next generation of software

    Dependencies
    ============

with TYPES; use TYPES;
with IMPORTANCE;
with SSI_DRIVER;


------------------
package SSI_OUT is
------------------


   procedure PUT(COMMAND : UINT16_ARRAY; SUCCESS : out BOOLEAN);



      where:

      COMMAND is the DPU command to be sent via the SSI interface.
      LEVEL   determines at what priority.

  procedure RESET renames SSI_DRIVER.RESET;


        performs a reset of the SSI interface and is identical to a call
        to the RESET procedure in package SSI_DRIVER
```

#### 6.4.2.62 ssi_out.adb

```
Extracted from file "ssi_out.adb"


 Function
 ========

 This file contains the body of package SSI_OUT.
 It provides routines to send data to the DPU via the SSI.

-----------------------
package body SSI_OUT is
   -----------------------


    The following is a routine internal to the package

   -----------------------
   function PUT_AND_CHECK(COMMAND : UINT16_ARRAY) return BOOLEAN;
   -----------------------


    where COMMAND contains the DPU command to be transmitted to the DPU.
    Any error will cause this routine to issue a message and reset the software.

    Create an instance (SSI_PORT) of a mutex semaphore using package MUTEX.

   SSI_PORT : MUTEX.SEMAPHORE;


   -----------------------------------------------------------------
   procedure PUT(COMMAND : UINT16_ARRAY; SUCCESS : out BOOLEAN) is
      -----------------------------------------------------------------


       seize the SSI for writing using SSI_PORT.SEIZE

       send the supplied command to the ssi_driver code using PUT_AND_CHECK.

       release the SSI for writing by using SSI_PORT.RELEASE.

   ----------------------------------------------------------------
   function PUT_AND_CHECK(COMMAND : UINT16_ARRAY) return BOOLEAN is
      ----------------------------------------------------------------


       write the SSI block to the DPU using SSI_DRIVER.PUT

       check the returned error code

          if the error code is OK (i.e. 0) then return true

       else if there was an error (error code < 0)

          send an exception report packet with the error ...

          ... and reset the SSI (software reset---no hardware reset) using RESET

          then return false indicating an error

       Otherwise

          Return FALSE indicating and error
```

#### 6.4.2.63 task_report.ads

Extracted from file "task_report.ads"

```
    Function
    ========

    This file contains the specification for package TASK_REPORT.

    The function of this package is to provide routine(s) to construct and
    place Task Parameter Report packets into the telemetry queue
    prior to their being transmitted to the ground.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

----------------------
package TASK_REPORT is
----------------------


  procedure PUT(TID      : UBYTE;
                FID      : UBYTE;
                PARAMS   : UINT16_ARRAY;
                SIZE     : INTEGER);


    The procedure PUT constructs and places a Task Parameter Report packet
    associated with TID and FID
    in the telemetry queue. The interface is as follows:

    where:


    PARAMS   specifies an array of parameters to be loaded into the packet.
             Note - the index range of the parameter array should start at 0.

    SIZE     specifies the number of parameters to be loaded from PARAMS.

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER);


    The procedure LOAD stores the parameters associated with
    TID and FID in a standard area. This location is checked if a request
    is made to dump those parameters at a later time.

    The interface is as follows:

    where:


    PARAMS   specifies an array of parameters to be loaded associated with
             TID and FID.
             Note - the index range of the parameter array should start at 0.

    SIZE     specifies the number of parameters to be loaded from PARAMS.

    NOTE: Alternative 'flavours' of this command now follow:

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16);


  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16;
                 PARAM2   : UINT16);

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16;
                 PARAM2   : UINT16;
                 PARAM3   : UINT16);
```

These LOAD procedures store 1,2 or 3 parameters respectivley
associated with TID and FID in a standard area.
This location is checked if a request
is made to dump those parameters at a later time.

The interface is as follows:

where:

```
PARAMS   specifies an array of parameters to be loaded associated with
         TID and FID.
         Note - the index range of the parameter array should start at 0.

SIZE     specifies the number of parameters to be loaded from PARAMS.
```

```
function  SEND(TID                   : UBYTE;
               FID                   : UBYTE;
               SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN ;
```

The FUNCTION  SEND constructs and places a Task Param Report containing
the parameters associated with TID and FID saved in the
standard area by the various 'flavours' of LOAD.

Returns TRUE if parameters found and send

#### *6.4.2.64 task_report.adb*

```
Extracted from file "task_report.adb"


 Function
 ========

 This file contains the body for package TASK_REPORT.

 The function of this package is to provide routine(s) to construct and
 place Task Parameter Report packets into the telemetry queue
 prior to their being transmitted to the ground.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------
package body TASK_REPORT is
-------------------


    Create a table of valid TID/FID combinations, how many expected prameters
    and default the location they will be stored to zero.

    Set up an area to store the parameters in.

   procedure PUT(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER) is


       Flag presence or absence of CRC in data field header

       Calculate and load packet length

       Load TID, FID and supplied parameters into packet

       Attempt to put packet record into queue using TMQ.PUT.

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAMS   : UINT16_ARRAY;
                 SIZE     : INTEGER) is


      Loop over the table of valid TID/FID combinations.

         If it knows about this TID/FID and the size
         (i.e. the number of parameters) is correct

            If this is the first time these params have been stored

               Set up the location to store them

            Copy parameters into table at specified location

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16) is


      Perform a call to the general purpose LOAD routine
      with 1 parameter

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16;
                 PARAM2   : UINT16) is


      Perform a call to the general purpose LOAD routine
      with 2 parameters.

  procedure LOAD(TID      : UBYTE;
                 FID      : UBYTE;
                 PARAM1   : UINT16;
                 PARAM2   : UINT16;
```

```
              PARAM3   : UINT16) is

    Perform a call to the general purpose LOAD routine
    with 3 parameters.

function SEND(TID                     : UBYTE;
              FID                     : UBYTE;
              SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN is


    Loop over the table of TID/FID combinations loaded
    so far.

       If this is a valid TID/FID combination
       and data has been stored

          Copy params into a packet and send it using PUT.

          Return a success condition.

    If no match was found with a previously supplied TID/FID
    combination, send an illegal parameters report packet.

    Return a failure condition.
```

***6.4.2.65 taskman.ads***

Extracted from file "taskman.ads"

```
    Function
    ========

    This package contains the specification for the TASKMAN package.
    The function of this package is to interpret the Task
    Management Telecommands and forward them to the appropriate code.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

------------------
package TASKMAN is
------------------


  function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN;


    The function REQUEST provides the means of passing the telecommand
    to the package for action.

    where:

    TC_PACKET contains the packet to be interpreted and executed.
```

#### *6.4.2.66 taskman.adb*

```
Extracted from file "taskman.adb"


 Function
 ========

 This package contains the body for the TASKMAN package.
 The function of this package is to interpret the Task
 Management Telecommands and forward them to the appropriate code.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-----------------------
package body TASKMAN is
-----------------------


  function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


     Set up default error condition of command not being accepted.

     Select action on the basis of packet subtype.

       When the packet subtype is Start Task...

         Set up default error of Illegal TID

         Select Action on the basis of the Task Identifier (TID)

           when TID is Blue Load Centroid Table

             Start the loading of the Blue Centroid Table

           when TID is Blue Load Window Table

             Start the loading of the Blue Window Table

           when TID is Blue Load DPU Deduced Window

             Do nothing is this is now always running.

           when TID is Blue Integration

             Start the Blue Detector Integration

           When TID is start the HV Ramp

             Start the HV ramp task

           when TID is Blue Camera Head Reset

             Reset the Blue Camera Head

           When the TID is a Move Filter Wheel Instruction

             Start moving the filter wheel

           When the TID is a Move Dichroic Instruction

             Start moving the dichroic

           When the TID is a contingency heater control

              Provided normal automatic heater control is disabled

                 Enable contingency heater control

               otherwise

                 Flag as an error with an unsuccessful acceptance packet.

                 Flag command as not accepted.

           When the TID is an automatic heater control
```

```
         Provided contingency heater control is disabled
            Enable automatic heater control
         Otherwise
            Flag as an error with an unsuccessful acceptance packet.
            Flag command as not accepted.
      When TID indicates a secondary voltage command
        Enable the secondary voltage.
      When the TID indicates DPU Science
        Start automatic 'handshake' of Science Data with DPU.
      When the TID indicates the DPU Heartbeat Watchdog.
        Ensure DPU Heartbeat watchdog monitor is started.
        Enable the Bent Pipe Diagnostic.
      When the TID indicates the DEMPSU
        Reset/Turn-on the DPU
      When the TID indicates an RBI Watchdog.
        Ensure the RBI Watchdog is started.
      When the TID indicates HK
        Ensure HK monitoring is enabled.
      When the TID indicates autonomous safing
        Ensure Autonomous Task is enabled
      When TID indicates ICB Direct Control
        Enable the ability to talk to the ICB directly.
      when TID is any other value
        flag as an invalid task command
    End of Selection.
  When the packet subtype is Stop Task...
    Prepare Default Error of illegal TID
    Select Action on the basis of the Task Identifier (TID)
      when TID is Blue Load Centroid Table
        Stop the loading of the Blue Centroid Table
      when TID is Blue Load Window Table
        Stop the loading of the Blue Window Table
      when TID is Blue Load DPU Deduced Window Table
        Flag as an invalid task command as no longer valid
      when TID is Blue Integration
        Stop the blue integration
      When TID indicates HV Ramp
         Stop the HV ramp task
      When TID indicates the Filter Wheel
        Stop moving the filter wheel
      When TID indicates Dichroic
        Stop moving the dichroic
```

```
      When TID indicates the contingency heater control
         Stop the contingency heater control
      When TID indicates the normal automatic heater control
         Stop the normal automatic heater control.
      When TID indicates the Secondary Voltages
        Disable the secondary voltages
      When TID indicate DPU science
        Disable the 'handshake' between the ICU and DPU of the
        science data.
      When TID indicates the DPU Heartbeat Watchdog
        Disable the DPU Heartbeat Watchdog.
      When TID indicate the DEMPSU
        Power down the DPU.
      When the TID indicates the RBI watchdog
        Disable the RBI watchdog.
      When TID indicates Housekeeping
        Disable the HK.
      When the TID indicates autonomous safing
        Disable the Autonomous Safing Task
      When the TID indicates the ICB DIRECT
        Disable the ability to write to the MACSbus ICB directly.
      when TID is any other value --------------------------------
        Flag is as an illegal task command.
    End of Selection
  When the packet subtype is Load Task...
    Set up a default Illegal FID error.
    Select Action on the basis of the Task Identifier (TID)
      when TID is Blue Load Centroid Table
        Load the centroid boundaries in the Blue system
      when TID is Blue Load Window Table
        Load the Window descriptions into the Blue system
      when TID is Blue Integration.
        Select action on the basis of the Function Identifier (FID)
          when FID is Blue Acquisition Mode
            Set Blue System Acquisition Mode
          when FID is Blue Double Threshold
            Set the Blue System Double Event Threshold
          when FID is Flood LED current
            Set the Flood LED current.
          when the FID is Enable Frame Tag
            If the frame tag value is zero
              Disable frame tags
```

```
      Otherwise
        Enable them.
    If FID ios Camera Running
      Set the Camera Running bit as per request
    when FID is any other value
      Flag as an illegal task command
  End of Selection
when TID is HV ramping
   Provided its the correct FID
      Load HV ramp parameters
   Otherwise
      Flag as an invalid task command.
When TID is Move Filter Wheel
  Select action on basis of FID
    If FID indicates a filter wheel movement parameter
      Load up the parameter
    When FID indicates the coarse sensor current
      Load up the coarse sensor current
    When FID indicates the fine sensor current
      Load up the fine sensor current
    When FID indicates the f/w step rate
      Load up the f/w step rate
    Any other FID value
      Flag as an invalid task command.
If the TID indicates a Move Dichroic
  Select action on the basic of the FID value
    When the FID indicate Dichroic direction/method
      Load Dichroic direction/method
    When the FID indicate Dichroic step rate
      Load Dichroic step rate.
    Any other value of FID
      Flag as an invalid task command.
If the TID indicate contingency heater control
   Provided its enabled
      Accept the command containg the heater configuration.
    otherwise
      Send an unsuccesful acceptance packet
      Flag as an error
When TID indicates normal automatic heater control
   Provided its enabled
      And its a valid FID
```

```
                  Load up the parameters
            Otherwise
                  Flag as invalid task command.
            otherwise
            Send an unsuccesful acceptance packet.
            Flag as an error.
      When TID indicates Direct ICB command
        Select action on value of FID
           when FID indicates a direct write to an ICB port
             and the option is enabled
                O/P datum to specified address and subaddress
             otherwise
                Flag as an error
          For any other value of FID
            Flag as an invalid task command.
      When TID indicates an RBI watchdog
          IF the FID is valid
            Load up the watchdog parameters
            If those parameters are not accepted.
               Send the appropriate unsuccesful acceptance packet
          All other FID's
            Flag as invalid task command
      when TID is DPU Direct
        Send parameters in the packet as a direct command to the DPU
      when TID is any other value
        Flag as an invalid task command
    End of Selection
  When the packet subtype is Report Task...
      If it's a valid read ICB port type
         and its enabled
            Request the task report and flag as accepted
         otherwise
            Flag as an error
      All other FIDs
        Send a normal task report packet using TASK_REPORT.SEND.
  When the packet subtype is Mode Transition...
    Send parameters to the MODEMAN.TO_MODE
End of Selection

If command was flagged as an invalid task management command,
inform the ground

Return success only if we had both a valid task command and
the command was not rejected by the functions called.
```

#### *6.4.2.67 tc_q.ads*

Extracted from file "tc_q.ads"

```
    Function
    ========

    This file contains the specification for the package TC_Q. It
    supplies the routines that manipulate the telecommand queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010
    The OBDH protocol is defined in XM-IF-DOR-0002

---------------
package TC_Q is
---------------


      Define number of slots NO_SLOTS in Telecommand Queue

      Define telecommand queue data structure as follows

            Description                  Size (Words)
            ===========                  ============

      ********************************
      *      Packet Slot 0           *       124
      *------------------------------*
      *    ... and so on until...    *       124
      *------------------------------*
      *      Packet Slot n-1         *       124
      ********************************

       Two pointers are used to indicate the 'occupation' of the queue.

       The Input Pointer indicates the packet slot into which the
       the next packet will be written.

       The Output Pointer indicates the packet slot from which the
       the next packet should be taken.

       In addition, there is a communication area (CCA) which the spacecraft examines
       to determine the location of a TM packet to be collected or into which
       a TC packet should be loaded.

       *************************************
       *     RBI Status Word               *
       *-----------------------------------*
       * Start Address of TM Source Packet *
       *-----------------------------------*
       *     Length of TM Source Packet    *
       *-----------------------------------*
       * Start Address of TC Source Packet *
       *************************************

      Create instance of Q data structure, and fix at location in memory (determined from MEMLOC).

       Define the input and output pointers at a fixed location in memory and zero them.

  procedure RESET;


      This procedure resets (i.e. clears) the TC queue

  procedure REMOVE(PCKT : in out PACKET.TC_TYPE );


      This procedure removes a packet from the TC queue

      where:

      PCKT is the packet removed from the TC queue.

  procedure ADD;


      This procedure informs the ICU that the s/c had DMAd a TC packet
```

```
    NOTE: This routine is now obsolete and should be removed. Its function
    is now handled by a low level assembler routine in package RBI_IH.

function IS_EMPTY return BOOLEAN;


    This function determines whether the TC queue is empty
    It returns TRUE if the queue is empty

function IS_FULL return BOOLEAN;


    This function determines whether the TC queue is full.
    It returns TRUE if the queue is full
```

***6.4.2.68 tc_q.adb***

```
Extracted from file "tc_q.adb"


 Function
 ========

 This file contains the body for the package TC_Q. It
 supplies the routines that manipulate the telecommand queue directly.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010.
 The OBDH protocol is defined in XM-IF-DOR-0002

--------------------
package body TC_Q is
--------------------


   Define telecommand queue data structure as follows (this information repeated for
   convenience from the specification).

         Description                 Size (Words)
         ===========                 ============

   ********************************
   *      Packet Slot 0           *       124
   *------------------------------*
   *    ... and so on until...    *       124
   *------------------------------*
   *      Packet Slot n-1         *       124
   ********************************

    Two pointers are used to indicicate the 'occupation' of the queue.

    The Input Pointer indicates the packet slot into which the
    the next packet will be written.

    The Output Pointer indicates the packet slot from which the
    the next packet should be taken.

    In addition, there is a communication area which the spacecraft examines
    to determine the location of a TM packet to be collected or into which
    a TC packet should be loaded.

    *************************************
    *     RBI Status Word               *
    *-----------------------------------*
    * Start Address of TM Source Packet *
    *-----------------------------------*
    *     Length of TM Source Packet    *
    *-----------------------------------*
    * Start Address of TC Source Packet *
    *************************************

   procedure RESET is


       Set the start and end pointers to the location of 1st packet.

       Store the Start address of the 1st packet in the comm area
       using RBI.SET_COMM_AREA_TC_INFO.

       Inform s/c we are ready to receive a packet by setting the
       appropriate RBI status word bit using RBI.SET_TC_READY.

   procedure REMOVE(PCKT : in out PACKET.TC_TYPE ) is


       Copy packet from current slot specified by the output pointer into PCKT.

       calc next output pointer value, watching for 'wraparound'

       Inform s/c we are ready to receive a packet again by setting the
       appropriate RBI status word bit (provided the queue is not full)
       using RBI.SET_TC_READY.

   procedure ADD is
```

```
    NOTE: This routine is now obsolete and should be removed.
    Its function is now handled by a low level assembler routine in
    package RBI_IH.

   Tell s/c we can't receive TC packets using RBI.SET_TC_READY.

    Packet has already been stored by s/c
    So calculate next slot index indicated by the value of the input pointer
    and watching for 'wraparound'

    Now set up new address for next packet using RBI.SET_COMM_AREA_TC_INFO

    Now tell s/c we can accept TC packets again if q not full using RBI.SET_TC_READY.

function IS_EMPTY return BOOLEAN is


    Return TRUE if Input Pointer equals the Output Pointer

    Otherwise return FALSE

function IS_FULL return BOOLEAN is


    Calc value of input pointer of next (after current) packet slot to be written.

    Return TRUE if it is the same as the output pointer.
```

#### *6.4.2.69 tc_verify.ads*

Extracted from file "tc_verify.ads"

```
    Function
    ========

    This file contains the specification for the TC_VERIFY package.

    That package supplies the routines that construct and send the
    telecommand verification packets.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

--------------------
package TC_VERIFY is
--------------------


  procedure SUCCESSFUL_ACCEPTANCE
                 (TC_SEQ_COUNT_AND_SRC: UINT16);


      This procedure constructs and sends a successful telecommand acceptance
        packet to the telemetry queue.

      where:

      TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                           telecommand being verified.

  procedure UNSUCCESSFUL_ACCEPTANCE
                 (TC_SEQ_COUNT_AND_SRC: UINT16;
                  ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                  NO_PARAMS           : UINT16;
                  PARAMS              : UINT16_ARRAY);


      This procedure constructs and sends an unsuccessful telecommand
      acceptance packet to the telemetry queue.

      where:

      TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                           telecommand being verified.

      ERROR_CODE           specifies the reason for failure

      PARAMS               specify any parameters associated with the
                           error code (NOTE - unlike other routines in the
                           ICU code, the first index of this array must be 1)


  procedure UNSUCCESSFUL_EXECUTION
                 (TC_SEQ_COUNT_AND_SRC: UINT16;
                  ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                  NO_PARAMS           : UINT16;
                  PARAMS              : UINT16_ARRAY);


      This procedure constructs and sends an unsuccessful telecommand
      execution packet to the telemetry queue.

      where:

      TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                           telecommand being verified.

      ERROR_CODE           specifies the reason for failure

      PARAMS               specify any parameters associated with the
                           error code (NOTE - unlike other routine in the
                           ICU code, the first index of this array must be 1)

  procedure  REPORT_ERROR(ERROR : PACKET.COMMAND_ERROR_TYPE;
                         TC_SEQ_COUNT_AND_SRC: UINT16);
```

This is a simplified version of UNSUCCESSFUL_ACCEPTANCE,
for use when there are no parameters.

#### ***6.4.2.70 tc_verify.adb***

Extracted from file "tc_verify.adb"

```
 Function
 ========

 This file contains the body for the TC_VERIFY package.

 That package supplies the routines that construct and send the
 telecommand verification packets.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

-------------------------
package body TC_VERIFY is
   -------------------------


    The specification for this package's internal routine follows:
    ==============================================================

   procedure UNSUCCESSFUL(
                       SUB_TYPE            : PACKET.TELEMETRY_SUBTYPE;
                       TC_SEQ_COUNT_AND_SRC: UINT16;
                       ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                       NO_PARAMS           : UINT16;
                       PARAMS              : UINT16_ARRAY);


    where:

    SUB_TYPE             is the packet sub-type being output
                         (unsuccessful acceptance or execution).

    TC_SEQ_COUNT_AND_SRC is the sequence count and source flag of the
                         telecommand being verified.

    ERROR_CODE           specifies the reason for failure

    NO_PARAMS            specifies how many params are supplied

    PARAMS               specify any parameters associated with the
                         error code

    The body for this package's internal routine follows:
    =====================================================

   procedure UNSUCCESSFUL(
                       SUB_TYPE            : PACKET.TELEMETRY_SUBTYPE;
                       TC_SEQ_COUNT_AND_SRC: UINT16;
                       ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
                       NO_PARAMS           : UINT16;
                       PARAMS              : UINT16_ARRAY) is


       Create instance of verification packet of requested sub-type

       Return as successful with no further action if an internal command is causing the error
       (as this will have no source and sequence count parameter - the 'impossible' value
       of FFFF (hex) is used to indicate this).

       Get the time and place it in packet using TIME_MAN.TIME_STAMP.

       Flag CRC as present

       Store the number of parameters supplied

       Calculate and load packet length

       Copy originating sequence count and source flag into packet

       Copy error code into packet

       and then copy in the associated parameters

       Place packet in queue using TMQ.PUT.
```

```
 The bodies for this package's externally visible follow:
 ========================================================

procedure UNSUCCESSFUL_EXECUTION
  (TC_SEQ_COUNT_AND_SRC: UINT16;
   ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
   NO_PARAMS           : UINT16;
   PARAMS              : UINT16_ARRAY) is


    Call UNSUCCESSFUL with sub-type specifying Unsuccessful Execution

procedure UNSUCCESSFUL_ACCEPTANCE
  (TC_SEQ_COUNT_AND_SRC: UINT16;
   ERROR_CODE          : PACKET.COMMAND_ERROR_TYPE;
   NO_PARAMS           : UINT16;
   PARAMS              : UINT16_ARRAY) is


    Call UNSUCCESSFUL with SUB_TYPE specifying Unsuccessful Acceptance

procedure SUCCESSFUL_ACCEPTANCE
  (TC_SEQ_COUNT_AND_SRC: UINT16) is


    Create verification packet of sub-type Succesful Acceptance

    Return as successful with no further action
    if an ICU internal command (i.e. if source and sequence count is set
    to the immpossible value of FFFF hex) caused the error

    Get the time and place it in packet using TIME_MAN.TIME_STAMP.

    Flag CRC as present

    Calculate and load packet length

    Copy originating sequence count and source flag into packet

    Place packet in queue using TMQ.PUT.

procedure REPORT_ERROR(ERROR : PACKET.COMMAND_ERROR_TYPE;
                       TC_SEQ_COUNT_AND_SRC: UINT16) is


    If the error code is in the unsuccessful execution range

       Call UNSUCCESSFUL_EXECUTION with ERROR supplied and
       number of parameters set to zero.

    Otherwise

       Call UNSUCCESSFUL_ACCEPTANCE with ERROR supplied and
       number of parameters set to zero.
```

#### *6.4.2.71 tcq.ads*

Extracted from file "tcq.ads"

```
    Function
    ========

    This file contains the specification for the package TCQ.
    That package supplies the low level routines that manipulate the
    telecommand queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010.
    The OBDH protocol is defined in XM-IF-DOR-0002.

--------------
package TCQ is
--------------


  procedure RESET;


    This procedure resets (i.e. clears) the telecommand queue


  procedure GET(PCK         : in out PACKET.TC_TYPE;
                GOOD_PACKET : out BOOLEAN);


    This procedure returns the next valid telecomand packet received
      to the caller.

    where:

    PCK   is the returned packet.

    GOOD_PACKET - always returns TRUE.

  procedure ADD renames TC_Q.ADD;


    The procedure is called when an EOTC Instruction to User
    interrupt is received (i.e. that a TC packet has been added to the
    TC queue).
    NOTE: This routine is now obsolete and should be removed. Its function is
    now handled by a low level assembler routine in package RBI_IH.
```

### *6.4.2.72 tcq.adb*

```
Extracted from file "tcq.adb"


 This package body implements the specification given in TCQ.ADS

 Dependencies
 ============

with TC_Q;
with TMQ;
with TC_VERIFY;
with TYPES; use TYPES;
with CRC;
with HK;
with SYSTEM;
with MEMLOC;


-------------------
package body TCQ is
   -------------------


    Data Global to this package
    ===========================

    As this package only returns valid packets, it requires a table
    of valid types and subtype, and the associated error conditions,
    as follows:

        Subtype 0 1 2 3 4 5 *  Comments
    Type
      1         ? ? ? ? ? ? ?
      2         I o o I I I I  Device Commanding
      3         ? ? ? ? ? ? ?
      4         ? ? ? ? ? ? ?
      5         I o o o o o I  Task Management
      6         I o o o I I I  Memory Maintenance
      7         ? ? ? ? ? ? ?
      8         ? ? ? ? ? ? ?
      9         I o I o o o I  Telemetry Maintenance
     10         I I o o I o I  Time Management
     11         ? ? ? ? ? ? ?
     12         ? ? ? ? ? ? ?
     13         I o I I I I I  Test Commands
     14         ? ? ? ? ? ? ?
     15         ? ? ? ? ? ? ?

    where:

    o = valid type/subtype, i = invalid subtype, ? = invalid type

    The specification and body for the internal routine follow:

    where:

    TC_PACKET is the packet to be checked for validity.

   function VALID_PACKET(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


       Assume by default we have a good packet.

       If a good packet

          Perform Valid APID check

             If not, note and flag it as a bad packet as invalid APID.

       If still a good packet

          Perform Packet Length Check (i.e. is it in a valid range)

             If not, note and flag it as a bad packet with invalid length.

       If still a good packet

          and a CRC is flagged as being present

             Perform CRC check
```

```
        If the CRC check fails

           Note and flag it as a bad packet with incorrect checksum.

  If still thought to be OK

     Look up error condition, if any, as a function of packet type
     and subtype, from the table described above.

     Select next action on the basis of the value returned.

        If packet OK

           Flag it is a good packet.

        If an invalid packet is present

           Determine whether because it is a bad type or bad sub-type.

           Load up the packet type and subtype into the parameter
           array for the error packet to be sent.

           Finally flag as bad packet.

  If it's not a good packet so far

     Construct and place Unsuccessful Acceptance
     Telemetry Packet in the telemetry queue with the appropriate error code.

     Increment bad packet count HK.TC_BAD for HK purposes.

  Return whether it was a good (TRUE) or bad (FALSE) packet.

procedure RESET is


  Perform queue reset by calling TC_Q.RESET

procedure GET(PCK         : in out PACKET.TC_TYPE;
              GOOD_PACKET : out BOOLEAN) is


  Commence infinite loop

     If the telecommand queue is empty

        then wait a while

     otherwise

        Remove a packet from the queue using TC_Q.REMOVE.

        If function VALID_PACKET returns a value of TRUE
        (i.e. we have a valid packet).

           then exit from this loop (and therefore procedure), indicating success.

     End Loop

 Package TCQ Code
 ================

 Perform a failsafe Reset Queue on Package Elaboration using RESET.
```

#### 6.4.2.73 time_man.ads

Extracted from file "time_man.ads"

```
    Function
    ========

    The file contains the specification for the Time Manager Package TIME_MAN.
    This package, together with the package BCP4_IH, supplies routines to
    support On-Board Time Management.

-------------------
package TIME_MAN is
-------------------


  function REQUEST(TC_PACKET : PACKET.TC_TYPE)
                  return BOOLEAN;


      This routine implements the On-Board Time Management Packets TC(10,x)
      contained in TC_PACKET. The format of these packets is defined in
      the Packet Structure Definition document PX-RS-0032. Of those, only
      the following are required to be supported.

      TC(10,2) - Enable Time Synchronization.
      TC(10,3) - Add Time Code.
      TC(10,5) - Enable Time Verification.

      In this release, the function always returns TRUE.

  function VERIFICATION_ACTIVE
    return BOOLEAN;


      This function returns TRUE if the process of verifying the time
      is in progress.

  function SYNCHRONISATION_ACTIVE
    return BOOLEAN;


      This function returns TRUE if the process of synchronizing the time
      is in progress.

  function TIME_STAMP
    return PACKET.TIME_TYPE;


      This function returns the current on-board time in a format suitable
      for direct insertion into a packet.
      (see the RBI package for details of the format).

  function OBT_AT_NEXT_BCP4
    return RBI.OBT_TYPE;


      This function
      1) waits until the next BCP4 pulse from the spacecraft
      2) returns the On-board time at that pulse in the format as
         provided by the RBI
         (see the RBI package for details of the format).
```

#### *6.4.2.74 time_man.adb*

Extracted from file "time_man.adb"

```
 Function
 ========

 The file contains the body for the Time Manager Package TIME_MAN.
 This package, together with the package BCP4_IH, supplies routines to
 support On-Board Time Management.

-------------------------
package body TIME_MAN is
-------------------------


    The following is the specification for a task internal to this package.
    It constructs and sends and enable time verification package after the
    initial processing of the BCP4 interrupt by package bcp4_ih.

   task BCP4 is
      entry START;
   end BCP4;


   function REQUEST(TC_PACKET : PACKET.TC_TYPE) return BOOLEAN is


       Determine action on the basis of the packet sub-type.

         If we have received a Time Synchronisation Packet

           Inform world that we are synchronising by setting
           the appropriate flag for use in HK.

           Enable time synchronisation by commanding the
           RBI configuration register appropriately using RBI.SET_SYNC_READY

         If we have received an Add Time Code Packet

           Remember the most significant byte from the time information
           supplied by the packet.

           Copy remaining significant 4 bytes into work array

           Convert them to RBI OBT (On-Board Time) format and
           load into RBI registers using RBI.SET_OBT

           Now disable Time synchronisation by commanding the RBI
           configuration register accordingly using RBI.SET_SYNC_READY.

           Now update DPU time to agree with the new time value using the special
           version of the IC_SYNCH_CLK with the length set to zero.

           Finally, tell world we are no longer synchronising by resetting
           the appropriate flag in HK.

         If we have received an Enable Time Verification Packet

           Inform world we are verifying the time by setting the
           appropriate flag for HK

           Start BCP4 processing task by calling BCP4.START.

           and leave it to do the work

         For any other packet sub-types.

           Do nothing.

       In this release, always return success.

   task body BCP4 is


       Begin infinite loop

          Wait until a call to start the task occurs i.e. BCP4.START

          Wait for the next BCP4 and get the corrected RBI format OBT
          using the OBT_AT_NEXT_BCP4 function.
```

```
        Create instance of a Time Management Report packet.

        Now build Time Verification Packet

        Flag CRC as present

        Calculate and load packet length.

        Construct Most Sig Byte of time stamp from value
        extracted from Add Time Code packet and held in memory.

        Construct remaining bytes from corrected OBT.

        And send it to to TM queue using TMQ.PUT.

      and disable BCP4 processing

      and inform world via HK we have finished verifying the time.

function SYNCHRONISATION_ACTIVE return BOOLEAN is


    Return the value of the synchronising flag

function VERIFICATION_ACTIVE return BOOLEAN is


    Return the value of the verification flag

function TIME_STAMP return PACKET.TIME_TYPE is


    Construct Most Sig Byte of time stamp from value extracted
    earlier from the Add Time Code packet and held in memory

    Get current corrected On-Board Time from the RBI using RBI.CORRECTED_OBT.

    Construct remaining bytes of time stamp from it;

    Return the time stamp.

function OBT_AT_NEXT_BCP4 return RBI.OBT_TYPE is


    Enable BCP4 processing at interrupt level by setting a flag the assembler
    code in bcp4_ih will poll at the next BCP4 interrupt.

    then wait for bcp4 int to be processed by code in
    package bcp4_ih (i.e. load up the OBT).

    Correct and return the On Board Time from the RBI using RBI.CORRECT_OBT.
```

#### *6.4.2.75 timer_a_ih.ads*

Extracted from file "timer_a_ih.ads"

```
    A block of variables are declared in MECHANISMS as part of the
    specification of that package
    so that they are 'visible' to this package which actually
    performs the movement. It is compiled separately as it is run
    at interrupt level and therefore a different set of compilation flags
    must be used.

    Enables the phase coils for the stepper motor driving DEVICE
    (FILTER_WHEEL or DICHROIC) as specified by the bit pattern contained
    in PHASE (1 = enabled) as follows:

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------


function FW_PHASE return UINT16;


    Returns a bit pattern specified by earlier calls to SET_PHASE
    commanding the filter wheel stepper motor for which the bit pattern
    PHASE was non zero. As before, the bits are defined as follows
    ( 1 = enabled)

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------

function DM_PHASE return UINT16;


    Returns a bit pattern specified by earlier calls to SET_PHASE
    commanding the dichroic stepper motor for which the bit pattern
    PHASE was non zero. As before, the bits are defined as follows
    ( 1 = enabled)

                                     L.S.B.
    -----------------------------------------
    | Phase 1 | Phase 2 | Phase 3 | Phase 4 |
    -----------------------------------------

function COARSE_POSITION_SENSED return BOOLEAN;


    Returns TRUE if the filter wheel coarse sensor is currently detected.

function FINE_POSITION_SENSED return BOOLEAN;


    Returns TRUE when the filter wheel fine position sensor is detected
```

#### ***6.4.2.76 timer_a_ih.adb***

Extracted from file "timer_a_ih.adb"

```
   procedure START(INIT_PULSE_RATE : UINT16) is


       Flag that the next pulse will be the first

       Init pulse counter

       Zero fine pulse sensor counter

       Set up first Timer A Interrupt

   procedure STOP is


       Cancel current interrupt

   procedure WHEN_ALARM_HAPPENS  is


       If the ICB is still busy at non-interrupt level

          Set up another timer A interrupt in a little while
          using ALARMCLOCK.SETALARM

          and return from interrupt

       Re-enable SSI and RBI interrupts so they can still be processed
       (as otherwise they are 'locked out' due to being lower priority

       Inform ICB Driver that it will be running at interrupt level

       Provided mechanisms are flagged as in use

          1st reset timer A ready for next pulse
            (rate dependent on requested movement speed
             set up in calls to the MECHANISMS package)

          If this is not the first pulse

             As the mechanisms will now have settled, we check for
              exit conditions resulting from prior pulse.

             If exit condition is when we reach the specified steps

                If we have reached the max steps requested,
                flag as finished

                If we are within braking distance

                   then start decelerating if cruising or accelerating

             If the exit is on seeing the coarse sensor

                Flag as finished when coarse sensor set

                Set error flag if we have gone beyond
                max requested steps

             If the exit is on seeing the next fine sensor

                If fine sensor seen

                Increment count of fine sensor pulses

                Flag as finished when pulse count at requested max pulses

                Set error flag if we have gone beyond
                max requested pulses

             If the exit is on seeing the coarse sensor and
             fine sensor pulse  (i.e. we are at datum)

                Can we can see the coarse sensor?

                Set finished flag if we also see the fine sensor

                If finished, reset filter wheel position
```

```
        If we can we see the coarse sensor
           Adjust speed to be fixed but very slow
        but if we can't see the coarse sensor
           Keep the speed fixed but at standard pull-in rate
        Set error flag if we have gone beyond max requested steps
     If exit is at dichroic positive excursion
        If we are moving to maximum excursion
           Flag as finished when steps >= 31 and phase is 1
           If we are moving n steps
           Flag as finished when we reach them
     If exit is at dichroic negative excursion
        If we are moving to max dichroic excursion
           Flag as finished when steps >= 31 and phase is 2
        but if we are moving n steps
           Flag as finished when we reach them
  Otherwise, if this is not the first pulse
     Flag as unfinished
     and reset first_pulse flag as false
  If the finished or error flag is set
     Terminate movement
  Otherwise, we have not finished so
     Determine next phase
     Send phase line commands to appropriate device via SET_PHASE
     Examine which mechanism we are moving
        If it's the f/w
           Flag it as moving
           Increment its position couter
        If it's the dichroic
           Adjust position counter according to movement direction
           Remember the last phase set for HK use
     Increment the pulse count;
     Now determine time interval for next pulse
     based on whether we are accelerating/deceleration etc
        If we are accelerating
           Increase pulse rate
           If now at max speed, switch to cruising
        If we are decelerating
           Calculate pulse rate downwards
           If now back to pull-in speed, switch to fixed speed
        if we are cruising
           Leave speed alone
        If the speed is fixed
```

```
                Do nothing

    Clear flag that we are at interrupt level

function SET_PHASE(DEVICE : in MECHANISM.DEVICE_TYPE;
                   PHASE : in UINT16
                   ) return BOOLEAN is


    It should be noted that the same TMSPU MACSbus sub address
    is used to command the stepper motor phases for both the
    filter wheel and dichroic as follows

      MSB
    -----------------------------------------
    | F1 | F2 | F3 | F4 | D1 | D2 | D3 | D4 |
    -----------------------------------------

    where D1->D4 are the dichroic motor phases.
          F1->F4 are the filter wheel motor phases.

    Determine which device is being commanded.

       If the filter wheel is being commanded

          Insert the requested phase bit pattern into the
          the appropriate part of the command word to be
          to be sent to the mechanisms.

          If it's a non zero phase, remember for recall
          as last active phase for the filter wheel.

       If it's the dichroic that's being commanded

          Insert the requested phase bit pattern into the
          the appropriate part of the command word to be
          to be sent to the mechanisms.

          If it's a non zero phase, remember for recall
          as last active phase for the dichroic.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus).

    Always return OK as the ICB routines inform the ground if there
    was an error.

function FW_PHASE return UINT16 is


    Return the last non zero phase pattern sent to the filter wheel.

function DM_PHASE return UINT16 is


    Return the last non zero phase pattern sent to the dichroic.

function COARSE_POSITION_SENSED return BOOLEAN is


    Get datum containing the value from the appropriate address
    on the MACSbus.

    The format of the datum now received is as follows:

    -------------------------------------------------
    |C0|C1|C2|C3|C4|C5|C6|C7|XX|XX|XX|XX|XX|XX|XX|CS|
    -------------------------------------------------

    where C0->C7 is the 'raw' current of the requested secondary circuit.
          XX is "don't care".
          CS is coarse sensor status, 1 = 'seen'.


    Extract sensor status from the CS field within the datum
    and return it.

function FINE_POSITION_SENSED return BOOLEAN is


   Get Data from the appropriate MUX port on the ICB MACSbus.
```

```
Extract and return the bit corresonding to the Fine Sensor status.
```

### *6.4.2.77 tm_man.ads*

Extracted from file "tm_man.ads"

```
    Function
    ========

    This file contains the specification for the telemetry manager package, TM_MAN.


    Reference
    =========

    XMM-OM/MSSL/ML/0010.1


-----------------
package TM_MAN is
-----------------


function REQUEST(TM_MAN_PACKET : PACKET.TC_TYPE) return BOOLEAN;


       This function provides the means of passing the telecommand
       to the package for further action.

       where :

       TM_MAN_PACKET contains the tc packet to be interpreted and executed.


function SID_STATUS(SID : PACKET.SID_TYPE) return BOOLEAN;


       This function reports on the TM packet generation status of a
       packet with the corresponding packet type specified by SID.

       where :

       SID is the tm packet sid to be reported

       If the generation of a TM packet with this SID is enabled then
       the function will return TRUE, FALSE otherwise.


function REPORT_STATUS(SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN;


       This procedure is responsible for generation of TM(9,1) packet in
       response to a TC(9,1) packet.

       where :

       SRC_AND_SEQUENCE_COUNT is the contents of the sequence count field
       of the associated telecommand.

       Returns TRUE if the command was successfully accepted


procedure VETO(TM_MAN_IGNORED: BOOLEAN);


    Ensures , if true, that STATUS always returns TRUE
```

#### *6.4.2.78 tm_man.adb*

Extracted from file "tm_man.adb"

```
 Function
 ========

 This file implements the body of the telemetry manager package, TM_MAN, for
 Operational mode.


 Reference
 =========

 XMM-OM/MSSL/ML/0010.1


----------------------
package body TM_MAN is
----------------------


    Define some package internal procedures

   function CHANGE_ALL(ENABLE_DISABLE         : BOOLEAN;
                       SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN;


    This internal  procedure changes the generation status of all applicable
    TM packets to that specified by ENABLE_DISABLE. The
    SEQUENCE_COUNT_AND_SRC parameter is needed in case of unsuccessful
    command execution

   function CHANGE_SPECIFIC(ENABLE_DISABLE         : BOOLEAN;
                            SID                    : PACKET.SID_RECORD_ARRAY;
                            SEQUENCE_COUNT_AND_SRC : UINT16;
                            PKT_LENGTH             : UINT16) return BOOLEAN;


    This internal procedure changes the generation status of the TM packets
    specified by the SID parameter to that specified by ENABLE_DISABLE.
    SEQUENCE_COUNT_AND_SRC parameter is needed in case of unsuccessful
    command execution



    Create the enabled array which contains true if a particular
    sid is to be enabled (ie a tm packet with that sid can be
    generated)



    Create the valid array which contains true if a particular
    sid is defined


  function REQUEST(TM_MAN_PACKET : PACKET.TC_TYPE) return BOOLEAN is


      Check whether CRC is present in tc packet

      Now determine packet subtype and act accordingly

         If 1 for a Report TM Packet Generation Status

            report the status using REPORT_STATUS

         If 2 for an enable Generation of all TM Packets

           Ignore as not supported by OM!

         If 3 for a Disable Generation of all TM Packets

            Disable all SIDs using CHANGE_ALL.

         4 for an Enable Generation of Specific Packets

            Enable a specific SID using CHANGE_SPECIFIC.

         5 for a Disable Generation of Specific Packets
```

```
         Disable a specific SID using CHANGE_SPECIFIC.

       Any other value, return false

function SID_STATUS(SID : PACKET.SID_TYPE) return BOOLEAN is


    Return the SID value in the valid sid array
    or'ed with the value in the enabled array

function REPORT_STATUS(SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN is


    Loop over the valid sid array, getting all SID enabled status
    and put them in an array

    Now create rest of the tm packet

    Put packet into tm queue using TMQ.PUT

function CHANGE_ALL(ENABLE_DISABLE         : BOOLEAN;
                    SEQUENCE_COUNT_AND_SRC : UINT16) return BOOLEAN is


    Loop over the enabled sid array

       Record enabled status in array

    Return success.

function CHANGE_SPECIFIC(ENABLE_DISABLE         : BOOLEAN;
                         SID                    : PACKET.SID_RECORD_ARRAY;
                         SEQUENCE_COUNT_AND_SRC : UINT16;
                         PKT_LENGTH             : UINT16) return BOOLEAN is


    Calculate number of sids to change

    If valid number of sids

       Set up error parameters just in case

       Test whether SID to change is a valid one

       If this is a valid SID

          Determine SID type is

             When fast HK

                If enabling this SID

                   If slow HK or science is enabled

                      then cannot enable fast HK

             When slow hk

                If enabling this SID

                   If fast HK is already enabled

                      then cannot enable slow HK

              When any science SID

                Determine whether this SID is already enabled

                If enabling this SID

                   If fast HK is already enabled

                      then cannot enable this science SID

                   Else

                      If SID already enabled

                         Do nothing

                   else

                         increment enabled science SIDs counter
```

```
                Else if disabling this SID
                   If SID already enabled
                      Then decrement science SID enabled counter
    Else set up error parameters
    If the SID status can be changed
       Record changed SID status
    Else
       Send unsuccessful acceptance packet using TC_VERIFY.UNSUCCESSFUL_ACCEPTANCE.
       Return FALSE
    Return TRUE.
procedure VETO(TM_MAN_IGNORED: BOOLEAN) is

    Set the override flag to supplied value.
```

#### *6.4.2.79 tm_q.ads*

Extracted from file "tm_q.ads"

```
    Function
    ========

    This file contains the specification for package TM_Q.

    That package supplies the low level routines that manipulate the
    telemetry queue directly.

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010
    The OBDH protocol is defined in XM-IF-DOR-0002

---------------
package TM_Q is
---------------

       Two pointers are used to indicate the 'occupation' of the queue.

       The Input Pointer indicates the packet slot into which the
       the next packet will be written.

       The Output Pointer indicates the packet slot from which the
       the next packet should be taken.

       Define the input and output pointers at a fixed location in memory.

  procedure RESET;

      This procedure resets (i.e. clears) the TM queue

  procedure ADD(PCKT : in PACKET.TM_TYPE );

      This procedure adds a packet to the TM queue

      where:

      PCKT is the packet to be added to the TM queue.

  function IS_FULL return BOOLEAN;

      This function determines whether the TM queue is full

      where IS_FULL returns TRUE if the queue is full

  procedure REMOVE;

      This procedure remove a packet from the telemetry queue after
      the s/c indicates it has taken a copy with an EOTM Instruction to User.

      NOTE: This routine should be removed as its function is now
      performed by a low-level assembler routine in package RBI_IH.

  function PACKET_COUNT return UINT16;

      Returns the current packet sequence count.
```

#### *6.4.2.80 tm_q.adb*

Extracted from file "tm_q.adb"

```
 Function
 ========

 This file contains the body for package TM_Q.

 That package supplies the low level routines that manipulate the
 telemetry queue directly.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010.

 The OBDH protocol is defined in XM-IF-DOR-0002

--------------------
package body TM_Q is
--------------------


    The telemetry queue is a area of memory defined as follows:

          Description                  Size (Words)
          ===========                  ============

    ********************************
    *      Packet Slot 0           *       259
    *------------------------------*
    *    ... and so on until...    *       259
    *------------------------------*
    *      Packet Slot n-1         *       259
    ********************************

    Two pointers are used to indicate the 'occupation' of the queue.

    The Input Pointer indicates the packet slot into which the
    the next packet will be written.

    The Output Pointer indicates the packet slot from which the
    the next packet should be taken.

    In addition, there is a communication area which the spacecraft examines
    to determine the location of a TM packet to be collected or into which
    a TC packet should be loaded.

    *************************************
    *     RBI Status Word               *
    *-----------------------------------*
    * Start Address of TM Source Packet *
    *-----------------------------------*
    *     Length of TM Source Packet    *
    *-----------------------------------*
    * Start Address of TC Source Packet *
    *************************************

    Create instance of Q data structure, and fix at location in memory

    Specify routines internal to this package
    =========================================

   function IS_EMPTY return BOOLEAN;


    returns TRUE if the telemetry Q is empty.

    Specify bodies for routines internal to this package
    ====================================================

   function IS_EMPTY return BOOLEAN is


       Return TRUE if Input Pointer equals Output Pointer.

       otherwise return FALSE.

    Specify bodies for routines visible externally
```

```
==============================================

procedure RESET is


    Set the start and end pointers to the 1st packet

    Ensure the s/c knows the queue is empty by using RBI.SET_TM_READY

    Reset the packet sequence counter to zero

procedure ADD(PCKT : in PACKET.TM_TYPE ) is


    If the queue is full (use IS_FULL fucntion)

       Raise a TM Q Overflow exception (This should never happen
         as TMQ package should guard against this?)

    Otherwise

       Store packet at next free slot

       Store sequence count in packet

       Prepare sequence count for next packet, performing 'wraparound'
         if necessary.

       If CRC required

          Convert packet to an array of 16 bit word

          Calc CRC location in words from
          packet length in bytes already in supplied packet

          Calculate CRC value using CRC.CALC_TM

          and place it at CRC location

       As we now manipulate items that may be manipulated/examined
       by an interrupt handler as well

       Grab them for exclusive use by blocking task pre-emption
       and interrupts by entering a critical section.

       Check here whether queue is now shown as empty (use IS EMPTY Function).
       If it is then the
       queue was empty prior to packet insertion.
       (Note: this is so because we haven't updated the pointers yet
       and so still reflect pre-insertion status.)

         If so, we need to inform s/c of the new packet address
         (derived from the Output Pointer) which is now available.
         Also tell the spacecraft its length.
         Note that the INPUT_POINTER = OUTPUT_POINTER at this stage.
         Use RBI.SET_COMM_AREA_TM_INFO to do this.

         Finally, ensure TM_READY bit is up using RBI.SET_TM_READY,
         to let spacecraft know that there are packets to take.

       Otherwise

         Do nothing, because there are still packets to be
         removed and therefore the spacecraft has the information
         it needs from a previous pass.

       Finally, calculate next slot index by incrementing the
       input pointer (and 'wrapping around' if necessary).

       Finally, allow manipulation by other code by leaving the critical section

procedure REMOVE is


    NOTE: This routine should be removed as its function is now
    performed by a low-level assembler routine in package RBI_IH.

    Ensure TM_READY bit is down while we process this

    Calculate new output index following packet removal

    If the queue is now empty
```

```
      Leave TM_READY bit low to inform s/c of the fact

   Otherwise

     set up packet information which enables the
     the spacecraft to fetch the next packet.

      Ensure TM_READY bit is up, to let s/c more packets to come

function IS_FULL return BOOLEAN is


   Calc Input Pointer of next (after current) packet slot to be written.

   Return TRUE if it is the same as the output pointer.

function PACKET_COUNT return UINT16 is


   Return the current sequence count.
```

#### *6.4.2.81 tmpsu.ads*

```
Extracted from file "tmpsu.ads"


    Function
    ========

    This file contains the specification for the TMPSU package. The package
    contains the software to control and monitor the Telescope Module Power
    Supply. It is based on document XMM-OM/IALS/SP/0002 -
    "TMPSU Electrical Specification".

----------------
package TMPSU is
----------------


  procedure SEND(
               SUBADR : in  SUB_ADDRESS_TYPE;
               DATUM  : in  UINT16;
               OK     : out BOOLEAN);


      Sends the data value DATUM to the MACS subaddress SUBADR of the
      TMSPU. OK is set to TRUE if no errors occur.

  procedure ACQUIRE(SUBADR : in  SUB_ADDRESS_TYPE;
                  DATUM  : out UINT16;
                  OK     : out BOOLEAN);


      Reads the data value DATUM from the MACS subaddress SUBADR of the
      TMSPU. OK is set to TRUE if no errors occur.

  function SET_SECONDARY_VOLTAGES(ON_OFF : BOOLEAN;
                                  SRC_AND_SEQUENCE_COUNT : UINT16)
                                  return BOOLEAN;


      Enables or disables (ON_OFF = TRUE or FALSE respectively) the secondary
      voltages that power the blue electronics.
      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

  function SECONDARY_VOLTAGES_ENABLED return BOOLEAN;


      Returns the status of the Secondary Voltages (TRUE = ON) for display
      in Housekeeping.

  function SET_COARSE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                              SRC_AND_SEQUENCE_COUNT : UINT16)
                                              return BOOLEAN;


      Sets the current for the coarse sensor illuminating LED in 'raw' units
      to be used when moving the filter wheel. The value is not used until
      a call to COARSE_SENSOR is made.
      SRC_AND_SEQUENCE_COUNT contains the sequence count field of the
      associated telecommand.

  function COARSE_SENSOR_CURRENT return UINT16;


      Returns the current for the coarse sensor illuminating LED in 'raw' units
      that is used when moving the filter wheel.

  procedure COARSE_SENSOR( ON_OFF : BOOLEAN);


      Turns on/off (ON_OFF = TRUE/FALSE) the illuminating LED used
      by the filter wheel coarse sensor. It uses the current specified in an
      earlier call to SET_COARSE_POSITION_SENSOR_CURRENT.

  function SET_HEATER_CONFIG(CONFIG : UINT16;
                             SRC_AND_SEQUENCE_COUNT : UINT16) return BOOLEAN;


      The bit pattern in CONFIG specifies which heater should be on or off
      ( 1 = on ) as follows:
                                              L.S.B.
```

| Temperature Control | | Focussing | |
|---|---|---|---|
| Main (HTR 1) | Forward (HTR 2) | Metering Rods (HTR 3) | Secondary Mirror (HTR 4) |

SRC_AND_SEQUENCE_COUNT contains the sequence count field of the associated telecommand.

function HEATER_CONFIG return UINT16;

Returns a bit pattern specifying the current heater configuration as follows:

L.S.B.

| Temperature Control | | Focussing | |
|---|---|---|---|
| Main (HTR 1) | Forward (HTR 2) | Metering Rods (HTR 3) | Secondary Mirror (HTR 4) |

function CURRENT(SECONDARY_VOLTAGE : UINT16) return UINT16;

Returns the current (in 'raw' units ) for the secondary supply circuit specified by SECONDARY_VOLTAGE as follows:

```
+25  V : 0
+15  V : 1
+11  V : 2
+5.3 V : 3
-5.3 V : 4
-15  V : 5
+28  V : 6
+ 5  V : 7
```

The values returned are used in the Housekeeping.

#### 6.4.2.82 tmpsu.adb

Extracted from file "tmpsu.adb"

```
 Function
 ========

 This file contains the body for the TMPSU package.  The package
 contains the software and data structures to control and monitor
 the Telescope Module Power
 Supply. It is based on document XMM-OM/IALS/SP/0002 -
 "TMPSU Electrical Specification".

package body TMPSU is


   procedure SEND(
                  SUBADR : in  SUB_ADDRESS_TYPE;
                  DATUM  : in  UINT16;
                  OK     : out BOOLEAN) is


       Send the DATUM to MACS sub-address SUBADR at the MACS address
       corresponding to the TMPSU on the Instrument Control Bus
       using ICB.PUT.

       Set OK to TRUE if no error occurs.

   procedure ACQUIRE(SUBADR : in  SUB_ADDRESS_TYPE;
                     DATUM  : out UINT16;
                     OK     : out BOOLEAN) is


       Gets the DATUM at MACS sub-address SUBADR at the MACS address
       corresponding to the TMPSU on the Instrument Control Bus
       using ICB.GET.

       Set OK to TRUE if no error occurs.

   function SET_SECONDARY_VOLTAGES(ON_OFF : BOOLEAN;
                                   SRC_AND_SEQUENCE_COUNT : UINT16)
                                   return BOOLEAN is


       Remember the last commanded secondary status.

       As the bit defining the status of the secondaries is
       combined with other bits, construct the bit pattern from the
       requested status of the secondaries and the last known values
       of the other bits.

       The layout is as follows:
       MSB
       -----------------------------
       |CS0|CS1|CS2|SC0|SC1|SC2|SE |
       -----------------------------

       where CS0->CS2 specify which secondary circuit is being monitored.
             SC0->SC1 specify the coarse sensor illuminating current.
             SE       specifies whether the secondaries are enabled.

       Write the bit pattern to the appropriate address & subaddress
       on the ICB (Macsbus) using ICB.PUT.

       Allow electronics to settle.

       If we had a macsbus error

          Restore record of current status to that of the last status noted earlier.

       Always return OK as the ICB routines inform the ground if there
       was an error via an error count in the HK.

   function SECONDARY_VOLTAGES_ENABLED return BOOLEAN is


       Return the stored status of the secondary supplies.

   function SET_COARSE_POSITION_SENSOR_CURRENT(CURRENT : UINT16;
                                               SRC_AND_SEQUENCE_COUNT : UINT16)
                                               return BOOLEAN is
```

```
    If the requested current is greater than the maximum  (7) allowed

       Reset it to the maximum allowed and note the value.

    else

       Simply note the value.

    Always return OK.

function COARSE_SENSOR_CURRENT return UINT16 is


    Return the stored 'raw' current to be used when powering the illuminating
    LED for the filter wheel coarse sensor.

procedure COARSE_SENSOR(ON_OFF : BOOLEAN) is


    If the LED is to be turned on

       Determine the current value from the earlier value( given by
       SET_COARSE_POSITION_SENSOR_CURRENT or a default value).

    else

       Use a value of zero.

    As the bits defining the 'raw' current to drive the illuminating
    LED of the filter wheel coarse sensor is combined with other bits,
    construct the bit pattern from the determined value of current and
    the last known values of the other bits.

    The layout is as follows:
    MSB
    -----------------------------
    |CS0|CS1|CS2|SC0|SC1|SC2|SE |
    -----------------------------

    where CS0->CS2 specify which secondary circuit is being monitored.
          SC0->SC1 specify the coarse sensor illuminating current.
          SE       specifies whether the secondaries are enabled.

    Write the bit pattern to the appropriate address & subaddress
    on the ICB (Macsbus) using icb_driver.put.

function SET_HEATER_CONFIG(CONFIG : UINT16;
                           SRC_AND_SEQUENCE_COUNT : UINT16)
                           return BOOLEAN is


    Loop over permitted heater configurations.

       If the request heater configuration is one of them

          Write the bit pattern to the appropriate address & subaddress
          on the ICB (Macsbus) using ICB.PUT.

          Remember the requested heater configuration for
          HK and heater control purposes.

          and exit with a success flag.

    Otherwise exit (in this release, also with a success flag).

function HEATER_CONFIG return UINT16 is


    Return the last commanded heater configuration.

function CURRENT(SECONDARY_VOLTAGE : UINT16) return UINT16 is


    If the requested circuit is outside the allowed range of circuits

       Return a zero.

    As the bits defining which secondary circuit is to be monitored are
    combined with other bits, construct the bit pattern from the
    requested secondary circuit and the last known values
    of the other bits.
```

```
The layout is as follows:
MSB
-----------------------------
|CS0|CS1|CS2|SC0|SC1|SC2|SE |
-----------------------------

where CS0->CS2 specify which secondary circuit is being monitored.
      SC0->SC1 specify the coarse sensor illuminating current.
      SE       specifies whether the secondaries are enabled.

Write the bit pattern to the appropriate address & subaddress
on the ICB (Macsbus) using ICB.PUT.

Wait for electronics to settle.

Write the bit pattern to the appropriate address & subaddress
on the ICB (Macsbus) to initiate an analogue to digital conversion
using ICB.PUT.

Wait a bit for the electonics to settle.

Get datum containing the value from the appropriate address
on the MACSbus using ICB.GET.

The format of the datum now received is as follows:

-------------------------------------------------
|C0|C1|C2|C3|C4|C5|C6|C7|XX|XX|XX|XX|XX|XX|XX|CS|
-------------------------------------------------

where C0->C7 is the 'raw' current of the requested secondary circuit.
      XX is "don't care".
      CS is coarse sensor status, 1 = 'seen'


Extract current value from the C0->C7 field within the datum
and return it.
```

#### 6.4.2.83 tmq.ads

Extracted from file "tmq.ads"

```
    Function
    ========

    This file contains the specification for the TMQ package.
    The function of that package is to provide routines to control
    access to the telemetry queue

    Reference
    =========

    The format of these packets is defined in the XMM-OM Telecommand and
    Telemetry Specification document XMM-OM/MSSL/ML/0010

    The protocol it implements is defined in the OBDH Bus Protocol Requirement
    Specification XM-IF-DOR-0002

--------------
package TMQ is
--------------


  procedure RESET;


      The procedure RESET resets (i.e. clears) the telecommand queue

  procedure REMOVE;


      The procedure REMOVE is called upon receipt of an EOTM Instruction to
      User from the spacecraft. This indicates that a TM packet has been
      taken

      NOTE: This routine should be removed as its function is now
      performed by a low-level assembler routine in package RBI_IH.

  procedure PUT(PCK : in PACKET.TM_TYPE);


      The procedure PUT places a packet in the telemetry queue

        where:

        PCK     is the packet to be inserted into the queue.


  function PACKET_COUNT return UINT16
    renames TM_Q.PACKET_COUNT;


      Rename (for convenience) the PACKET_COUNT function of package TM_Q.


  procedure SAFING(SAFING_VALUE : in BOOLEAN);


      Enables/disables (SAFING_VALUE = TRUE/FALSE) the automatic safing
      that takes place if TM queue becomes full.
```

#### *6.4.2.84 tmq.adb*

```
Extracted from file "tmq.adb"


 Function
 ========

 This file contains the body for the TMQ package.
 The function of that package is to provide routines to control
 access to the telemetry queue. It, in turn, call lower level routine in
 package TM_Q.

 Reference
 =========

 The format of these packets is defined in the XMM-OM Telecommand and
 Telemetry Specification document XMM-OM/MSSL/ML/0010

 The protocol it implements is defined in the OBDH Bus Protocol Requirement
 Specification XM-IF-DOR-0002

-------------------
package body TMQ is
-------------------


   Create Semaphore TM_QUEUE using package MUTEX.

   The specifications for this package's internal routine follow:

  procedure SEND_TO_TM_Q (PCK : in PACKET.TM_TYPE);


   where:

   PCK is the packet to be inserted into the queue.

  procedure SEND_TO_TM_Q (PCK : in PACKET.TM_TYPE)  is


     Commence infinite loop

       If the telemetry queue is full (use TM_Q.FULL)

         Wait a bit

         Increment a timout counter

         If we have now spent the timout period waiting for the TM queue
         to be become non-full

            If autonomous safing enabled

               If we are not already safing the instrument

                  and we are not already (full or intermediate) safed.

                     Initiate the intermediate safing of the instrument
                     using MODEMAN.TO_MODE.

               but if we have already started the safing process

                  Determine whether the safing process has finished.

            Reset the timout counter.

       Otherwise

         Reset the timout counter.

         Place packet in queue (via TM_Q.ADD).

         Exit from loop

       end infinite loop

  procedure PUT(PCK : in PACKET.TM_TYPE)  is


      Ensure we have exclusive use of the telemetry queue
      by use of the TM_QUEUE.SEIZE semaphore.
```

```
    Send the packet to the telemetry queue (via SEND_TO_TM_Q)

    Release the telemetry queue for use by other routines using TM_QUEUE.RELEASE.

procedure RESET is


   Call the reset routine in TM_Q for the telemetry queue

procedure REMOVE is


   Call the 'remove packet' routine for the telemetry queue.

   NOTE: This routine should be removed as its function is now
   performed by a low-level assembler routine.

procedure SAFING(SAFING_VALUE : in BOOLEAN) is


   Save requested autonomous safing status
```

#### 6.4.2.85 types.ads

Extracted from file "types.ads"

```
Function
========

The function of this package specification is to define the basic data
types used throughout the ICU ADA code.

Definitions
===========

Define Unsigned Byte type UBYTE

Define Signed Byte type BYTE

Define Unsigned 16 bit integer type UINT16

Define Signed 16 bit integer type INT16

Define Signed 32 bit type INT32

Define Unsigned Byte Unconstrained Array type UBYTE_ARRAY

Define Signed Byte Unconstrained Array type BYTE_ARRAY

Define Unsigned 16 bit Integer Unconstrained Array type UINT16_ARRAY

Define Signed 16 Bit Integer Unconstrained Array type INT16_ARRAY

Define Unsigned Nibble type

Define Unsigned Nibble Array Type

Define single bit Integer Unconstrained Array type BIT_ARRAY
```

***6.4.2.86 USERDEFS.asm***

**File is USERDEFS.asm**

```
  IF 0    ;~TI configuration constants now defined in linker control file
          DEFINE
          PREEMPTER_MASK
          DEFINE
          CONNECT_MASK
          DEFINE  ART_MASK
          DEFINE
          ARTTASK_MASK
          DEFINE
          ARTELAB_MASK
          DEFINE
          STACKTOP
          DEFINE
          MAINSTKTOP
          DEFINE
          MAINSTKSIZE
;
;
          Interrupt Masks
;
; Ada allows the connection of interrupts to task entries by use
; of "FOR task.entry USE intnumber".  The mask below indicates which
; 1750 hardware interrupts the user can DIRECTLY connect to with such
; a statement.  Note that ALL such interrupts, and any indirectly connected
; interrupts must also appear in PREEMPTER_MASK below.
CONNECT_MASK   EQU
          001BF
;
; The interrupt mask used during the execution of normal (post-elaboration)
; code, for both the main program and tasks, is defined below.
; Floating underflow must be disabled, Floating overflow, Fixed overflow
; and Timer B must be enabled (to validate).
ARTTASK_MASK   EQU
          0FDFF
;
; The mask used during the elaboration of the program, before the main
; program is started.  This is by default identical to the above.
ARTELAB_MASK   EQU
          0FDFF
;
; The masked used when runtime code is executing.  This must have in addition
; Fixed overflow disabled.
ART_MASK  EQU  0F5FF

; Next definition is a mask that also masks off any interrupts that might
; cause a task to be rescheduled
PREEMPTER_MASK EQU
          0FDFF-CONNECT_MASK-040
          ; Timer B too



;
;
          Stack Allocation
;
; Root initializes 2 stacks on startup.  Data space is laid out as follows:
;
;  STACKTOP
          =>   +---------------+
;
               |
               |
;
               |   Interrupt
          |
;
               |    Stack
          |
;
               |
               |
;  MAINSTKTOP
          =>   +---------------+
;
               |
               |
```

```
;
                |     Main
          |
;
                |    Stack
          |
;
                |
                |
;  MAINSTKTOP-MAINSTKSIZE =>
          +---------------+
;
                |
                |
;
                .
                .
;
                .     Free
          .
;
                .
                .
;
                |
                |
;  END_OF_DATA
          =>   +---------------+
;
                |
                |
;
                |    Static
          |
;
                |     Data
          |
;
                |
          &     |
;
                |     Code
          |
;
                |
                |
;  Low Mem ( 200 HEX)
          =>   +---------------+
;
;
STACKTOP  EQU  0FFFF
MAINSTKTOP     EQU
          0FD00
               ; Main stack top
MAINSTKSIZE    EQU
          01400
               ; Main stack size
  ENDIF   ; ~TI

;
; Time base constants.
;
; For the most part these constants are fixed by the tick rate of timer B
; (10KHZ in MIL-STD-1750).  Users who wish to adjust the shortest delay
; should be aware that values less than 2 are dangerous since the start
; of the delay is not necessarily synchronized with the clock tick.
;
; BEWARE!  These constants MUST appear in the order shown!!!
;
TIMES     CSECT,C
               ;~TI

          DEFINE
          ART#DURSML
ART#DURSML     EQU
          $
; conversion from TICKS to DURATION'SMALL (= 1.6384) as 3 word float
; this is the original
;         DATA 068DB,08B01,0AC71
; this is half the delay of the original
;         DATA 068DB,08B02,0AC71
; this is double the original
;         DATA 068DB,08B00,0AC71
```

```
; this is what we had
;         DATA 053a7,01102,05161
; this is what we want
          DATA    053e2,0d602,0238e
; shortest non-zero delay time allowed (in ticks)
          DATA 0,20
; ticks in one day
          DEFINE
          ARTONEDAY
ARTONEDAY EQU  $

; this is the original
;         DATA 0337F,09800

; this is what we want
          DATA 0202f,0bf00
; ticks in two days
; this is the original
;         DATA 066FF,03000
; this is what we want
          DATA    0405f,07e00
          END
```