Science Thread Module Specification

James Francis

MIT 64-30112.0601

Rev. D

April 29, 1993

Contents

Figures

Tables

1 Introduction

1.1 Scope

This document describes the Science Thread Module (SCIENCE), which is responsible for controlling the acquistion and processing of data generated by two sets of X-ray detectors, a Proportional Counter Array (PCA) and an All-Sky-Monitor (ASM). The Science Thread is executed by the 80286 (CPU).

This document provides a top level description of the Science Thread Module. There are a set of submodule specifications which describe the specific Science Modes. These documents rely heavily on this document (see Section 1.2).

This document does not describe the overall design of the EDS, nor does it describe the operating details of each component in this module. It assumes that the reader is extremely familiar with the Flight Software Global Design, with the "C" programming language and with the architecture of the EDS.

This document should be of primary interest to Flight Software Engineering, Flight Hardware Engineering, Ground Software Engineering and Software Quality Assurance.

1.2 References

MIT 64-02006.01 Rev. C .... "XTE EDS Hardware Functional Description and Requirements"
MIT 64-02009 Rev. D .... "EDS-PCA Science Requirements"
MIT 64-02012 Rev. C .... "EDS-ASM Science Requirements"
MIT 64-30102 Rev. E....................... "Flight Software Global Design".
MIT 64-30110.0401 Rev. D .... "Downlink Module Specification" (DOWNLINK).
MIT 64-30110.0701 Rev. C .... "Timing Module Specification" (TIMING).
MIT 64-30110.0901 Rev. C .... "Front Registers Module Specification" (FREG).
MIT 64-30110.1201 Rev. D .... "Interrupt Manager Module Specification" (INTR).
MIT 64-30110.1701 Rev. D .... "Thread Manager Module Specification" (THREAD).
MIT 64-30110.1901 Rev. D .... "Uplink Module Specification" (UPLINK).
MIT 64-30110.2001 Rev. C .... "Utility Module Specification" (UTIL)
MIT 64-30110.2301 Rev. D .... "Startup Module Specification" (STARTUP).
MIT 64-30110.2501 Rev. B .... "Rotation Utility Module Specification" (ROTATION).
MIT 64-30111.0201 Rev. C .... "Buffer Pool Manager Module Specification" (BUFP).

MIT 64-30111.0401 Rev. C .... "Partition Map Builder Module Specification" (PARTMAP).
MIT 64-30111.0501 Rev. C .... "Recoder Builder Module Specification" (RECODER).
MIT 64-30111.0801 Rev. C .... "Typer Builder Module Specification" (TYPER).
MIT 64-30112.0201 Rev. C .... "Housekeeper Thread Module Specification" (HOUSE).

The following is a list of all of the Science Mode Module Specifications which rely on this document:

MIT 64-30112.060101 Rev. C .... "Binned Mode Module Specification" (BINNED).
MIT 64-30112.060102 Rev. C .... "Pulsar Fold Mode Module Specification" (PULSAR).
MIT 64-30112.060103 Rev. C .... "Event Encoded Mode Module Specification" (EVENT).
MIT 64-30112.060104 Rev. C .... "Single Bit Mode Module Specification" (SINGLEBIT).
MIT 64-30112.060106 Rev. C .... "Burst Catcher Mode Module Specification" (CATCHER).
MIT 64-30112.060107 Rev. C .... "FFT Mode Module Specification" (FFT).
MIT 64-30112.060108 Rev. C .... "Position Histogram Mode Module Specification" (POSITION).
MIT 64-30112.060109 Rev. C .... "Multiple Time Series Mode Module Specification" (TIMESERIES).
MIT 64-30112.060110 Rev. C .... "Event by Event Mode Module Specification" (ASMEVENT).

1.3 Notations

This document assumes the following notations:

INT8 .... Refers to an 8-bit signed value.
UINT8 .... Refers to an 8-bit unsigned value.
INT16 .... Refers to a 16-bit signed value.
UINT16 .... Refers to a 16-bit unsigned value.
INT32 .... Refers to a 32-bit signed value.
UINT32 .... Refers to a 32-bit unsigned value.
BOOL .... Refers to a 16-bit value which represents a TRUE or FALSE condition.

1.4 Module Perspective

The following diagram is a copy of the Level 1 data flow diagram from the **Flight Software Global Design**, Section 4.1.2. This module's scope is reflected by the shaded areas:

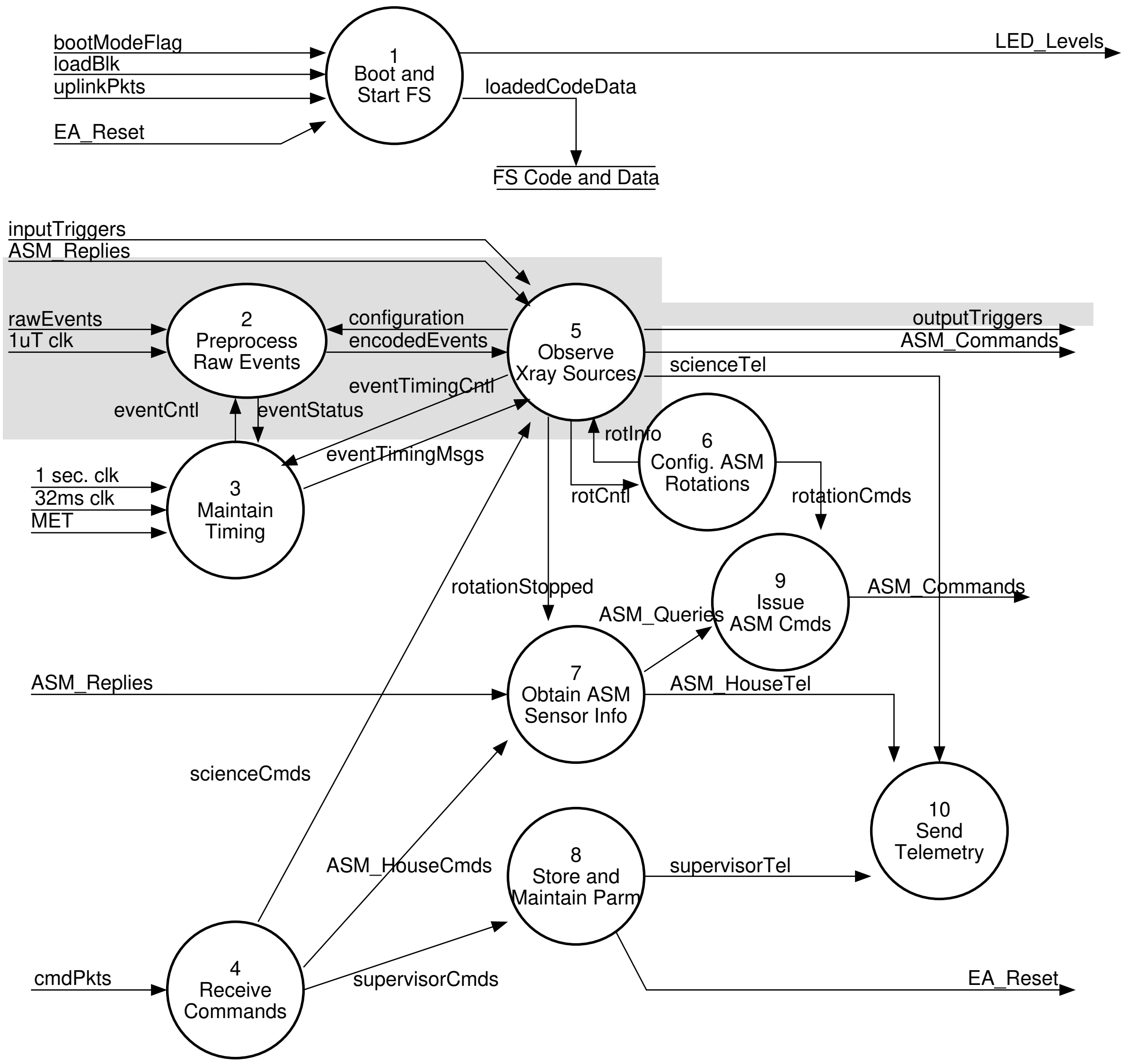

Figure 1 Module Perspective

1.5 Acronyms

In addition to the acronyms described in the **Flight Software Global Design Specification**, Section 1.4, this document defines and uses the following:

ASM-EA ...... All-Sky Monitor Event Analyzer
ms .... Millisecond (0.001 seconds)
PCA-EA .... Proportional Counter Array Event Analyzer
sec. .... Second
uT .... Microtick

1.6 Document Overview

Sections 1 and 2 provide an overview of the Science Thread Module, summarizing the purpose, services, context, and interfaces of the module. Section 3 describes the format of PCA and ASM event words, and all software structures and constants visible to modules outside of the Science Thread Module. Section 4 describes the commands interpreted by the Science Thread and the acknowledgments generated in response to the commands. Sections 5 and 6 describe the interrupts and events used by the Science Thread. Section 7 describes the initialization function provided to the STARTUP module, and describes the template for all science mode specific functions. Section 7 also describes the set of utility functions provided by the Science Thread Module for use by the various science modes. Section 8 describes some of the implementation details of the module. Section 9 describes which requirements are met by the module. Section 10 describes what types of tests need to be performed to ensure that the implementation of the module is operating properly. Appendix A.1 lists items which are currently undefined, or that need to be reviewed. Appendix A.2 illustrates a procedure which might be used when installing new science modes after launch.

2 Overview

2.1 Purpose

The purpose of the Science Thread is to control the acquisition and processing of PCA and ASM data and to package and post the processed data to be telemetered to the ground.

2.2 Services

The Science Thread Module provides the following services:

1) Control Event Processing	Configure and control the event processing circuitry during an observation, based on ground specified parameters.
2) Control ASM Rotations	Configure and synchronize ASM rotations based on ground specified parameters.
2) Pack Science Data	Pack all science data into telemetry packets, based on ground specified parameters, to minimize the telemetry usage.
3) Telemeter Science Data	Post packed science telemetry packets to be transferred to the SDS.
4) Detect Xray Bursts	Analyze obtained science data for xray bursts, based on ground specified parameters.

2.3 Module Context

The following illustrates the Science Thread Module relationships with the rest of the Flight Software residing in an EA:

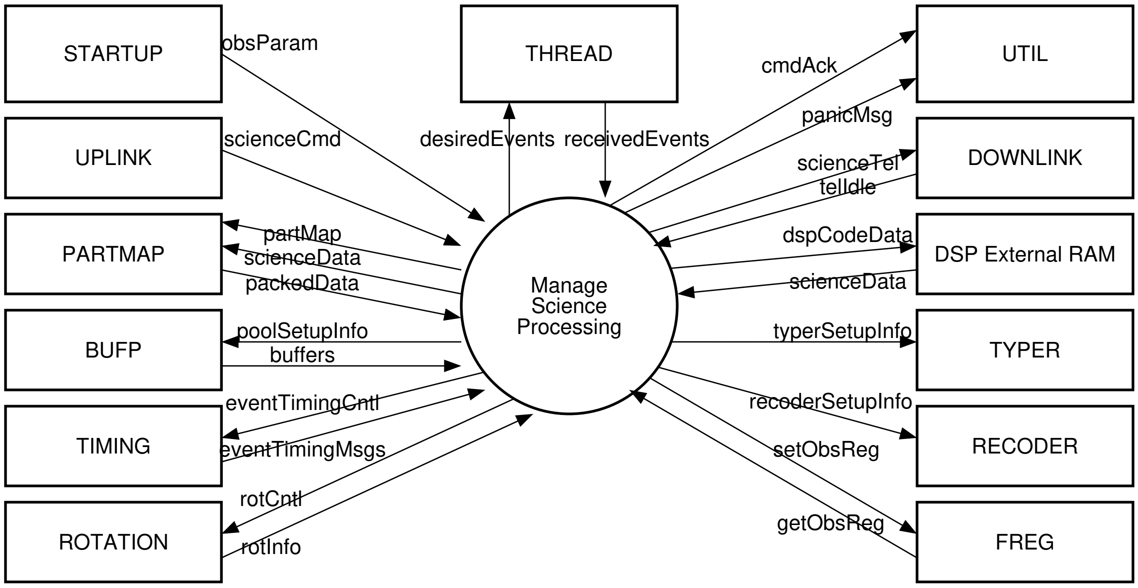

Figure 2 Science Thread Module Context Diagram

The Science Thread uses the THREAD module (see **Thread Manager Module Specification**) to relinquish control by waiting for events (desiredEvents) or yielding, and to obtain the desired events which were received (receivedEvents) once control is returned.

The Science Thread uses the STARTUP module (see **Startup Module Specification**) to obtain the desired science mode and parameters (obsParam). The Science Thread uses the UPLINK module (see **Uplink Module Specification**) to obtain run time command packets (scienceCmd), and issues command acknowledgments (cmdAck) using the UTIL module (see **Utility Module Specification**).

When configuring an observation, the Science Thread configures the PARTMAP module (see **Partition Map Module Specification**) by passing a partition map (partMap). PARTMAP uses this table to pack science data (scienceData, packedData). The Science Thread uses the TYPER module (see **Typer Builder Module Specification**) to unpack encoded typer table parameters (typerSetupInfo) into the PCA's Typer Table, and the RECODER module (see **Recoder Builder Module Specification**) to unpack encoded recoder table parameters (recoderSetupInfo) into the PCA's Recoder Table. The Science Thread writes the DSP's code and data (dspCodeData) directly into the DSP's external RAM, which is memory mapped into the 80286's address space.

The Science Thread uses the TIMING module (see **Timing Module Specification**) to configure and control the event timing circuits (eventTimingCntl) during an observation. It also uses the TIMING module to obtain timing messages (eventTimingMsgs), such as MET time information, and timing function call-backs. The Science Thread uses the FREG module (see **Observation Registers Module Specification**) to read and write the various Observation registers (setObsReg, getObsReg). If the Science Thread is configured to perform ASM camera rotations, it uses the ROTATION module (see **Rotation Utility Module Specification**) to control the rotations (rotCntl) and to obtain rotation timing information (rotInfo).

The Science Thread performs telemetry by using the BUFP module (see **Buffer Pool Manager Module Specificiation**) to setup a pool of telemetry buffers (poolSetupInfo), and to obtain buffers from the pool (buffers). It uses the **Partition Map Module** to pack the data (scienceData), read directly from the DSP external RAM, into the obtained buffers (packedData). The packed telemetry buffers (scienceTel) are then sent to the DOWNLINK module (see **Downlink Module Specification**) to be telemetered out of the EDS. If the Science Thread is commanded to reset the EA upon termination of the observation, it uses the **Downlink Module** to provide an indicator when all telemetry from the observation has been transferred out of the EA (telIdle).

If the Science Thread suspects that the Event Analyzer's RAM is corrupted, or if the Science Thread is commanded to reset the EA upon termination of the observation, it uses the UTIL module (see **Utility Module Specification**) to post panic messages and reset the Event Analyzer.

2.4 Theory of Operation

This section provides a brief overview of the Science Thread. The Science Thread may be running on one of two types of hardware. The first handles data produced by the Proportional Counter Array (PCA). The second handles data produced by the All-Sky-Monitor (ASM). The operation of the thread varies across these two types of hardware. On each PCA event analyzer (PCA-EA), the Science Thread controls the Science Data processing according to one of six possible modes. On each ASM event analyzer (ASM-EA), it controls the data processing according to one of three possible modes. For one of the ASM-EA modes, the Science Thread also controls the camera rotations.

All Science Observations use some common concepts and definitions, which are as follows:

Microtick .... A Microtick is defined by a transition of the spacecraft-supplied high-frequency clock. The clock produces 2^{20} microticks per second (roughly 1MHz).

Majortick .... A Majortick is defined by a transition of the spacecraft-supplied low-frequency clock. The clock produces 1 Majortick per second. There are exactly 2^{20} Microticks per Majortick.

Timebin... A Timebin is a variable unit of time during which all events detected are time-tagged with the same value. Timebin durations are specified in Microticks and are maintained by a Timebin Counter.

Interval... An Interval is a variable unit of time whose boundaries cause the Timebin Counter to reset to zero, and cause various status flags to be updated. Interval boundaries also cause an interrupt on the CPU (80286). Intervals may be synchronized to a Majortick. Their durations are specified in Microticks.

Partition .... A Partition is a single contiguous set of data, produced by the DSP, which is analyzed, packed, and telemetered by the Science Thread.

Partition Memory.............................. Partition Memory refers to the section of DSP RAM used to accumulate and process science data partitions.

2.4.1 PCA Operation

The following diagram illustrates the overall Science Data Flow in a PCA event analyzer:

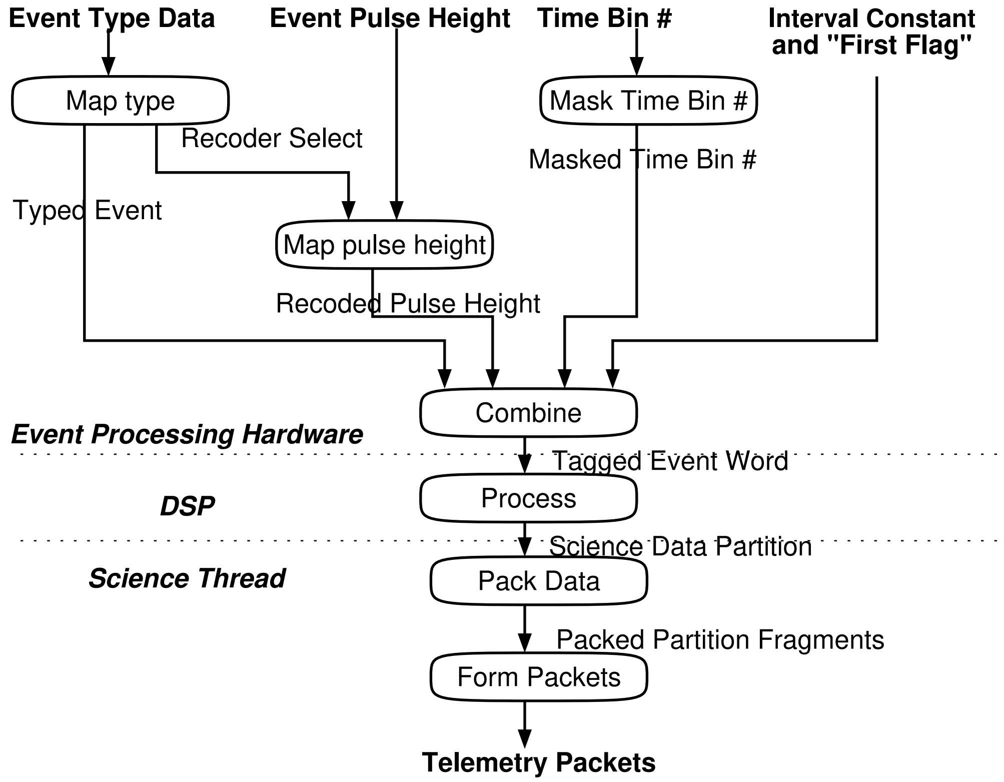

Figure 3 PCA Science Data Flow

The PCA event analyzer (PCA-EA) hardware contains event timing circuitry, an event type lookup table, an event pulse-height lookup table, and a Digital Signal Processor (DSP). This hardware is called the "Front End" of the PCA-EA. The hardware uses the timing circuitry and the lookup tables to generate time-tagged event words which are then processed by the DSP. The DSP accumulates the processed data into partitions. The Science Thread copies the accumulated data out of the DSP's RAM, and, if required by the science mode, zeros the portion of the DSP's RAM which contained the copied data. The Science Thread then packs the obtained data into Science Telemetry Packets, and posts the packets to be transferred out of the EDS.

In addition to building Science Telemetry Packets, the Science Thread is responsible for setting up the "Front End" of the PCA-EA, using parameters stored in the EDS's memory. It is also responsible for controlling the timing circuitry and partition selection during the observation. In the case of Binned Mode, when searching for Xray bursts, the Science Thread is also responsible for analyzing the processed data and asserting the burst trigger signals when an X-ray burst is detected.

The Science Threads, running on each PCA-EA, may govern the data processing according to one of the following modes:

* Binned .... Builds histograms of time-tagged event words
* Pulsar Fold .... Builds "folded" histograms of time-tagged event words
* Event Encoded .... Sequentially accumulates time-tagged event words.
* Single-Bit Code .... Builds a bit-map of Timebin and event occurrences.
* Burst Catcher .... Captures high-resolution image of X-ray bursts.
* FFT .... Peforms FFT's on Binned Mode histograms.

In addition to these modes, the Flight Software can be patched during flight to run additional modes (see Appendix A.2).

2.4.2 ASM Operation

The following diagram illustrates the overall Science Data Flow in an ASM event analyzer:

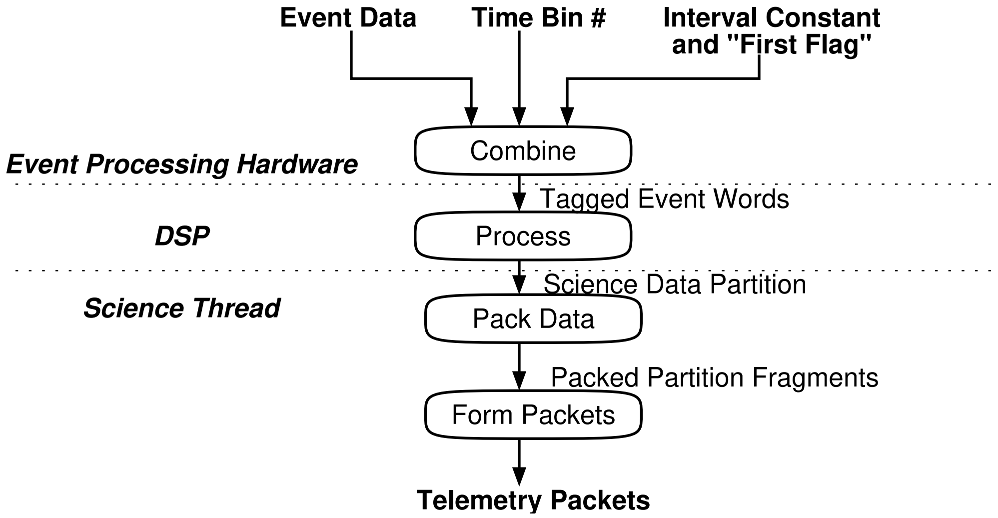

Figure 4 ASM Science Data Flow

The ASM event analyzer (ASM-EA) hardware consists of event timing circuitry, a Rotation Control Interface, an Engineering Sensor Interface, and a Digital Signal Processor (DSP). The Rotation Control Interface and the Engineering Sensor Interface share a communication channel. Their use must be coordinated so that the two interfaces are not used at the same time. This hardware is called the "Front End" of the ASM-EA.

On XTE, there are two event analyzers for the ASM. One event analyzer is responsible for managing the rotation of the ASM's cameras, acquiring the Engineering Sensor data, and acquiring Science Data that is only taken when the cameras are still (called a dwell). The other event analyzer accumulates Science Data in a fashion which does not require synchronization with the camera rotations and dwells.

On the ASM-EA responsible for camera rotation, only the Position-Histogram Science Mode may be run (the ground is responsible for ensuring that this requirement is met). In this mode, the Science Thread is responsible for configuring the DSP, configuring and controlling the interval timing circuitry (event timestamp values are ignored), rotating the cameras, acquiring data from the DSP and telemetering it to the ground. The Science Thread is also responsible for notifying the **ASM Housekeeper Thread** whenever the ASM Command/Housekeeping channel is available to acquire ASM Housekeeping information (see **Housekeeper Thread Module Specification**). Event acquisition is synchronized to the camera rotations, such that data is taken only when the cameras are still. During event processing, the hardware generates event words which are delivered to the DSP for further processing. The DSP reads event words from its Input FIFO. It then filters, processes and accumulates the processed data. The Science Thread reads the processed data directly from the DSP's RAM, packs the obtained data into Science Telemetry Packets, and posts the packets to be transferred out of the EDS. The Science Thread is also responsible for setting up the DSP, using parameters stored in the EDS's memory. The Science Thread running on the ASM-EA responsible for camera rotation may run the following mode:

* Position Histogram .... Anode Position histograms.

On the ASM-EA not responsible for camera rotation, the Science Thread controls both the DSP and the timing circuitry, using parameters stored in the EDS's memory. The hardware uses the timing circuitry to generate time-tagged event words which are delivered to the DSP for further processing. The DSP filters, processes and accumulates the processed data. The Science Thread reads the processed data directly from the DSP's RAM, packs the obtained data into Science Telemetry Packets, and posts the packets to be transferred out of the EDS. The Science Thread running on this EA may operate in one of two modes:

* Multiple-Time Series .... Time-series/Background/Pulse-Height histograms.

* Event-by-Event .... Sequentially accumulates ASM event words.

In addition to these modes, the Flight Software can be patched during flight to run additional modes (see Appendix A.2).

2.5 Interface Summary

The **Science Thread Module** uses the services provided by the following modules:

INTR Module .... Enables and disables interrupts for atomic operations.
THREAD Module .... Waits for Events. Provides Event Notification.
STARTUP Module .... Provides the Science Thread's Configured Observation Parameter Block
UPLINK Module............................. Provides Command Packets.
PARTMAP Module .... Builds Partition Map Entries for packing of Science Data. Packs Science Data using Partition Map Entries.
BUFP Module.................................. Provides Science Data Buffers.
DOWNLINK Module .... Posts telemetry packets for transfer out of the EA. Indicates when all telemetry from the observation has been transferred to the SDS.
RECODER Module .... Builds Recoder Lookup Table Entries (PCA Only).
TYPER Module............................... Builds Typer Lookup Table Entries (PCA Only).
TIMING Module............................. Installs Interval Interrupt Client Functions. Installs First-Flag Interrupt Client Functions. Calls installed client function on every Interval Interrupt. Calls installed client function on every First-Flag Interrupt. Controls the Interval Counter. Controls the Interval Constant. Controls the First-Flag Signal. Controls the Timebin duration. Enables and Disables Event Acquistion. Controls the Timebin Mask (PCA Only).
FREG Module .... Reads and writes Observation registers.
ROTATION Module .... Provides rotation information (ASM Only). Issues rotation commands (ASM Only).
UTIL Module .... Posts panic messages and resets the Event Analyzer. Performs fast memory copies. Performs 80286 address manipulations. Forms and posts command acknowledgements.
HOUSE Module .... Obtains/telemeters ASM Housekeeping Info (ASM Only).

BINNED Module............................. Performs the Binned Science Mode (PCA).
PULSAR Module .... Performs the Pulsar Fold Science Mode (PCA).
EVENT Module .... Performs the Event Encoded Science Mode (PCA).
SINGLEBIT Module .... Performs the Single Bit Science Mdoe (PCA).
CATCHER Module .... Performs the Burst Catcher Science Mode (PCA).
FFT Module..................................... Performs the FFT Science Mode (PCA).
POSITION Module......................... Performs the Position-Histogram Science Mode (ASM).
TIMESERIES Module .... Performs the Multiple Time Series Science Mode (ASM).
ASMEVENT Module .... Performs the Event by Event Science Mode (ASM).

2.6 Data Structures and Constants Summary

The Science Thread Module defines structures which are used by the ground to configure observations. It also defines its command and reply codes. The module uses definitions provided by the hardware and by the other modules which it uses.

2.7 Event Summary

The Science Thread may wait for the following events:

EV_NEW_UPLINK .... A new command packet is available for the thread.
EV_SCI_BUF_AVAIL .... A Science Telemetry Packet buffer is available.
EV_MAJOR .... A Majortick has just occurred.
EV_SCI_DATA_RDY .... A Science Data Set has been copied from the DSP's RAM.
EV_NO_MORE_DATA .... No more data will be produced for the remainder of an observation.

When performing Position-Histogram Mode, the Science Thread may issue the following event to the Housekeeper Thread:

EV_ASM_IF_AVAIL .... The ASM Command/Housekeeping Interface is available for use.

2.8 Interrupt Summary

The Science Thread Module installs two client functions using the **Timing Module**. The first client function is called whenever the Interval Timer causes an Interval Interrupt. The second client function is called whenever the DSP reads an Interval Marker word from its Input FIFO, and the First-Flag Interrupt is enabled for that interval (see **XTE EDS Hardware Functional Description and Requirements Specification**).

2.9 System Resource Utilization Summary

telemetry .... This module produces the bulk of the telemetry out of an EA. The ground must coordinate an EA's observation with all other EA's such that telemetry rates are not exceeded.
interval interrupt latency .... The interval interrupt client function, installed by this module, must not take a significant amount of time, since it will have an impact on all lower priority interrupts. If the interval duration is too short, relative to the execution time of the mode-specific client function, this handler could take too great a fraction of the cpu execution time.

first-flag interrupt latency .... The first-flag interrupt client function, installed by this module, must complete before the next data set of the observation has completed.

3 Data Structures and Constants

3.1 Hardware

This section provides an overview of the Event Analyzer hardware, with respect to the Science Thread. The hardware word descriptions shown in this section are for example only. For a detailed description of the Event Analyzer hardware, see the current **XTE EDS Hardware Functional Description and Requirements**.

3.1.1 Interval Marker Word

Whenever an Interval starts, the PCA and ASM event processing hardware write a single Interval Marker word into the DSP's Input FIFO. The format of PCA and ASM Interval Markers are identical and are as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	1	1	1	1	1	1	1	1	1	1	1	1	Int	Spil	E

E	Bit 0	If **F** is 0, 0 indicates Interval Marker, 1 indicates empty FIFO
Spil	Bit 1	0 indicates no spillage during previous interval. 1 indicates FIFO Spillage during previous interval
Int	Bit 2	0 indicates First Flag Interrupt disabled for Interval Marker 1 indicates First Flag Interrupt enabled for Interval Marker
1	Bits 3-14	Maps marker into high area of DSP's Scratch RAM
F	Bit 15	0 indicates Interval Marker or empty FIFO (see **E**). 1 indicates an event word

Figure 5 Interval Marker Word

If bit 15 (**F**) is one, then, on a PCA-EA, the word is a PCA event word (see Section 3.1.2), and, on an ASM-EA, the word is one of three ASM event words (see Section 3.1.3). If bit 15 (**F**) is zero, then the word is either an Interval Marker or an Empty FIFO flag word. In this case, if Bit 0 (**E**) is zero, then the word is an Interval Marker, and if one, an Empty FIFO flag.

3.1.2 Processed PCA Event Word

On the PCA-EA's, the event processing circuits time-tag and process PCA events to form a final event word. The meaning of the final output bits can vary, depending on how the timebin counter, timebin nibble mask, recoder table and typer table are configured. The following illustrates the various bit allocation ranges of the event word, as seen by the DSP:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F	d14	c13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	a2	a1	a0

IC

Typer

Recoder/Timebin

a0-a2	Bits 0-2	Usable by Timebin and Recoder Output.
b3-b12	Bits 3-12	Usable by Timebin, Recoder Output and Typer Output.
c13	Bit 13	Usable by Timebin, Recoder Output, Typer Output and Interval Constant.
d14	Bit 14	Usable by Timebin, Recoder Output and Interval Constant.
F	Bit 15	0 indicates Interval Marker. 1 indicates PCA event word

Figure 6 Processed PCA Event Word

The timebin generator can use a maximum of 15 bits, starting at bit 0 of the output word. Likewise, the recoder lookup table can use a maximum of 15 bits, starting at bit 0 of the output word. The typer lookup table can use a maximum of 11 bits, anywhere between bit 3 and bit 13, inclusive. The Interval Constant can use bits 13 and 14. Bit 15 is used by the hardware to indicate whether the word is a PCA Event word, or an Interval Marker (see Section 3.1.1).

For more detail, refer to the **Typer Builder Module Specification** and the **Recoder Builder Module Specification**.

3.1.3 ASM Event Words

The following illustrates the event words delivered by the ASM event processing hardware, as seen by the DSP:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word 1	F	1	IC		Pulse Height A											
Word 2	F	0	Det Id		Pulse Height B											
Word 3	F	0	Anode Code				Time Tag									

F	All words, Bit 15	0 if Interval Marker. 1 if ASM event word.
Pulse Height A	Word 1, Bits 0-11	Pulse Height Measured on "A" side of the anode
IC	Word 1, Bits 12-13	Interval Constant
Pulse Height B	Word 2, Bits 0-11	Pulse Height Measured on "B" side of the anode
Det Id	Word 2, Bits 12-13	Detector Id
Time Tag	Word 3, Bits 0-9	Time Stamp of Event
Anode Code	Word 3, Bits 10-13	Id of Measuring Anode, or event classification, where:
		0-7 *Identifies the measuring Anode*
		8 *Two or more anodes detected the event. No antico.*
		9 *One or more anodes detected the event, with antico.*
		10 *Antico activity only.*
		11-15 *Unused Codes*

Figure 7 ASM Event Words

3.2 Data Structures

3.2.1 Science Thread Command Packet

All Science Thread command packets consist of a CCSDS header, followed by a secondary header, followed by the thread id/command code byte, followed by a single reserved byte. The Science Thread Id and possible Science Thread command codes are listed in Section 3.3. A detailed description of the commands and their effects are described in Section 4.

The ground should pay particular attention to this section because it describes the meaning of the reply codes provided in the command acknowledgments corresponding to the commands.

For a description of the CCSDS and secondary headers, and of the format of the thread id/command code byte, refer to the **Flight Software Global Design Specification**, "Packet Formats : Command Packets."

3.2.2 Observation Parameter Block Formats

The following diagram provides an overview of all mode independent data needed to run an observation, where the arrowed lines refer to address pointers, and dashed boxes and lines refer to items which may or may not be present in a given configuration:

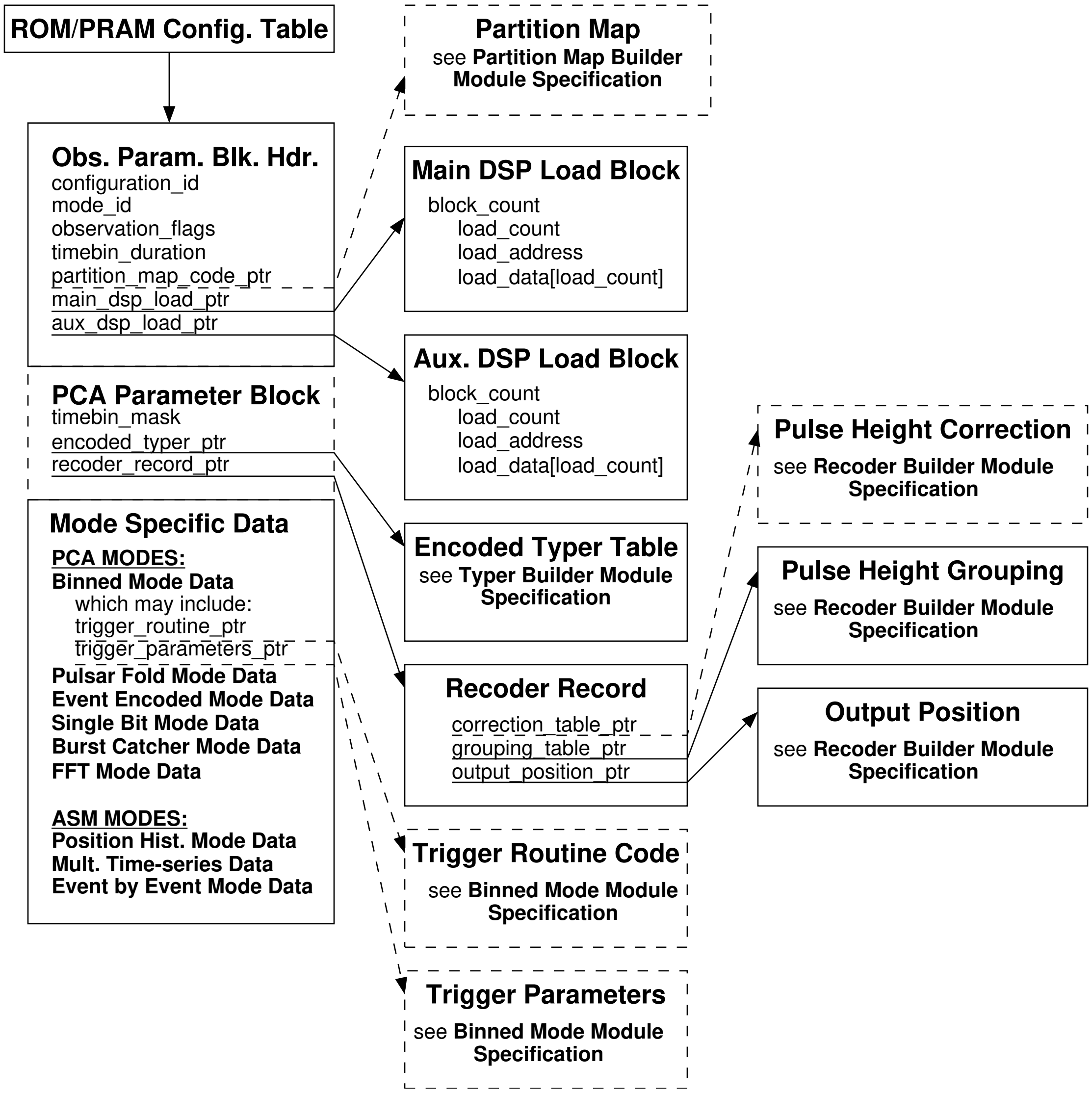

Figure 8 Observation Parameter Block Structure

Observation Parameter Blocks are obtained by the Science Thread when the thread starts, using the STARTUP module (see Section 2.3). Observation Parameter Blocks tell the Science Thread which mode to run, and provide the mode configuration parameters. Each Observation Parameter Block is associated with a single configuration and consists of the following components, some of which reference other configuration information; the Observation Parameter Block Header, if the mode is a PCA mode, a PCA Parameter Block, followed by a mode specific parameter block.

The Observation Parameter Block Header (see Section 3.2.2.1) consists of a configuration id, the id of the mode to run, various observation flags, a reference to the main DSP code and data to load (main DSP load block), a reference to the DSP parameters to load (auxiliary DSP load block), a Partition Map reference, and a Timebin duration.

The PCA Parameter Block (see Section 3.2.2.2) is only present if the mode id in the Observation Parameter Block Header indicates a PCA mode. It consists of the following; a Timebin Nibble Mask value, a pointer to an Encoded Typer Table, and a pointer to a Recoder Record (see Section 3.2.2.2.1). A Recoder Record contains three pointers, one to the Recoder Pulse Height Corrections Table, one to the Recoder Pulse Height Grouping Table, and one to the Recoder Output Position Table (see **Recoder Module Specification**).

The mode specific parameters are not described in this document, but, are described in the Science Mode documents (see Section 1.2). The type of mode specific parameters are determined by the Science Thread using the mode id in the Observation Header. The only mode specific parameter set which contains references to components outside of the main Observation Parameter Block is Binned Mode. In this case, the Binned mode parameters may point to a burst trigger routine, and a parameter block for use by the routine. Binned mode may use this routine and parameter block to examine the Binned Mode data for Xray burst activity.

3.2.2.1 Observation Parameter Block Header (22 bytes)

Every Observation Parameter Block starts with an Observation Parameter Block Header (shown in Figure 8). This header consists of the following:

Configuration Id (4 bytes) .... This value identifies the Observation Parameter Block. This value is telemetered to the ground with every data set generated during the observation.

Science Mode Id (1 byte)................. This value describes the Science Mode to run for the observation. It can take on one of the following values: PCA_MODE_BINNED, PCA_MODE_PULSAR, PCA_MODE_EVENT, PCA_MODE_SINGLEBIT, PCA_MODE_CATCHER, PCA_MODE_FFT, ASM_MODE_POSITION, ASM_MODE_TIMESER, ASM_MODE_EVENT.

[x,x,x,x,p,p,g,s] Obs. Flags (1 byte) This byte is available to contain various flags governing certain behaviors during the observation. These flags are:

x .... Unused (4 bits)

p .... Telemetry Priority (2 bits)
This two bit value selects which SDS telemetry priority to use for all parameter dump and science data sent during the observation.
If 0 use Real-Time Priority 0, if 1, use Priority 1, if 2, use Priority 2, and if 3, use Priority 3.

g .... Govern Telemetry Rate (1 bit)
If it is set to 1, telemeter data such that, at most, one science packet per MajorTick is sent to the SDS. If it is set to 0, attempt to send science packets at the rate at which they're produced.

s............... Suspend Telemetry (1 bit)
If this bit is set to 0, the specified Science Mode comes up in a resumed state, where Partition Header and Data packets are sent as they are produced. The mode remains in this state until commanded to suspend telemetry or terminate.
If this bit is set to 1, the specified Science Mode comes up in a suspended state, where Partition Data packets are discarded. The mode remains in this state until commanded to resume telemetry or terminate.

Timebin Duration (4 bytes) .... This value controls the time bin duration for all modes in the EA. This value is written into the Timebin Counter of both PCA and ASM Front End hardware.

Partition Map Addr. (4 bytes).......... This address points to a partition map which determines how to pack the science data into telemetry packets. Please refer to the **Partition Map Builder Module Specification** for a description of the format of a partition map.

Main DSP Load Block (4 bytes)...... This value points to the main DSP load block for the observation. The format of the referenced block is described in Section 3.2.2.1.1. Typcially, this block is used to load the main mode specific DSP code into the DSP's RAM.
Aux. DSP Load Block (4 bytes) ...... This value points to the Auxiliary DSP load block for the observation. The format of the referenced block is described in Section 3.2.2.1.1. Typcially, this block is used to load mode and observation specific parameters into the DSP's RAM.

3.2.2.1.1 DSP Load Block (between 8 and ~64K bytes)

A DSP Load Block structure is used for both Main and Auxiliary DSP Loads and consists of the following:

Block Count (2 bytes)....................... Specifies the number of DSP Blocks to load.
DSP Block [Block Count] .... The Block Count field is immediately followed by one or more DSP Blocks, where each block specifies a contiguous region of DSP memory to load.

Each DSP Block has the following format:

Address (4 bytes) .... Memory address to store the data. Although this value is in the 80286's address space, it should be chosen such that it maps into the DSP shared memory area. Therefore, the ground must ensure that this address is greater than or equal to MMAP_DSP_FLAT_START, and less than MMAP_DSP_FLAT_END, which are described in Section 3.3.
Word Count (2 bytes) .... Specifies the number of 16-bit words in the block
Data (Word Count * 2 bytes)............ The data to store in the DSP memory.

3.2.2.2 PCA Parameter Block (10 bytes)

Every Observation Parameter Block which selects one of the PCA Modes contains some data which is common to all PCA modes. This data is as follows:

Timebin Nibble Mask (1 byte) .... This value specifies the nibble mask to use for the timebin count. Only the lower 4 bits are used. Please refer to the **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.6.1.2, "Observation Control Register", element : TBMask, for a description of the effect of this value.
Reserved (1 byte)............................. This value is present to maintain alignment onto 16-bit words.

Encoded Typer Address (4 bytes) ... This is the memory location of an encoded Typer Table. The address is in segment:offset format. Refer to the **Typer Builder Module Specification** for a description of the format of an encoded table.

Recoder Record Address (4 bytes) .. This is the memory location of a Recoder Record. The address is in segment:offset format (see Section 3.2.2.2.1).

3.2.2.2.1 Recoder Record (12 bytes)

A Recoder Record consists of pointers to three Parameter Sets. The format of the Parameter Sets and any restrictions on the legal values for the following addresses are specified in the **Recoder Builder Module Specification**. The Recoder Record format is as follows:

Correction Parm. Address (4 bytes) . This specifies the address of the Pulse Height Correction Parameter Set.

Grouping Parm. Address (4 bytes) ... This specifies the address of the Pulse Height Grouping Parameters Set.

Output Parm. Address (4 bytes) .... This specifies the address of the Output Position Parameter Set.

3.2.3 Science Telemetry Data Formats

Each observation consists of four phases; a setup phase, a dump phase, a running phase, and a checking phase. During the setup phase, the Science Thread configures the DSP and hardware, and initializes all control variables.

During the dump phase, the Science Thread telemeters the parameter set for the observation and starts the observation. The format of the telemetered parameter set (Observation Parameter Dump) is described in Section 3.2.3.1.

During the observation's running phase, the Science Thread telemeters the data taken during the observation. Data telemetered from a given observation consists of a series of telemetry packets. The data from each observation is broken into a series of partitions, which are then further broken into telemetry packets. The first packet of a partition consists of a Partition Header followed by one or more Partition Data packets, which contain the partition's data. The format of the telemetered science data is described in Section 3.2.3.2.

During the checking phase, the Science Thread tests the integrity of the tables and parameters used to generate the science data. The results of these tests are telemetered as an Observation Coda telemetry packet. The format of an Observation Coda is described in Section 3.2.3.3.

3.2.3.1 Observation Parameter Dump

At the beginning of each observation, the parameters used by the observation are telemetered to the ground. All such telemetry packets are assigned a "*Data Stream Id*" of STREAM_ID_OBS_PARAM, which is defined in Section 3.3.2.2. Please refer to the **Flight Software Global Design Specification**, "Packet Formats : Telemetry Packets" for a description of the telemetry packet headers and "*Data Stream Id*". The Observation Parameter Dump consists of a series of Observation Components. Each component is sent using one or more telemetry packets. The first packet of a component is called a Component Header Packet, whose format is described in Section 3.2.3.1.1. Subsequent component packets are called Component Data Packets, and have the format described in Section 3.2.3.1.2. Component header and data packets are distinguished by the sequence number at the start of each packet. Header packets have a sequence number of 0, whereas data packets have a sequence number greater than 0. A list and description of the various type of components is in Section 3.2.3.1.3.

3.2.3.1.1 Component Header Packet Format

The format of the first packet of a component consists of the following:

Component Sequence # (1 byte)....... For a Component Header packet, this value must be zero. If it is not zero, then this is a Component Data Packet.

Component Tag (1 byte).................. This identifies the type of component being sent. This can have one of the following values (refer to Section 3.3 for a description of these codes):
OBS_COMP_HEADER,
OBS_COMP_MAIN_DSP,
OBS_COMP_AUX_DSP,
OBS_COMP_PARTMAP,
OBS_COMP_PCA,
OBS_COMP_TYPER,
OBS_COMP_RECODER,
OBS_COMP_REC_CORR,
OBS_COMP_REC_GROUP,
OBS_COMP_REC_POS,
OBS_COMP_BINNED,
OBS_COMP_TRIGGER,
OBS_COMP_PULSAR,
OBS_COMP_EVENT,
OBS_COMP_SINGLEBIT,
OBS_COMP_CATCHER,
OBS_COMP_FFT,
OBS_COMP_POSITION,
OBS_COMP_TIMESER,
OBS_COMP_ASMEVENT.

Configuration Id (4 bytes) .... This is a copy of the configuration id stored in the main Observation Header, from which this component was retrieved (see Section 3.2.2.1).
Component Address (4 bytes) .... This is the physical memory location of the component being telemetered.
Component Length (2 bytes) .... This is the length, in bytes, of the all of the component's data, including the data sent in subsequent Component Data Packets.
Component Data (N bytes) .... This is the first, and usually the only, section of data from the component. If the data cannot fit into the packet, subsequent Component Data Packets will be sent, containing the rest of the component's data.

3.2.3.1.2 Component Data Packet Format

The format of the subsequent packets of a component consists of the following:

Component Sequence # (1 byte)....... For a Component Data Packet, this value must be greater than zero. If it is zero, the it identifies the packet as a Component Header Packet. For each data packet sent, this value increments by one, starting at one.
Component Tag (1 byte)................... This identifies the type of component being sent.
Component Data (N bytes) .... This is a continuation of the data sent by the previous Component Header or Data Packet in the sequence. If the last Component Data Packet of a sequence contains an odd number of bytes, an extra pad byte is included, which is not reflected in the Component Length field of the Component Header Packet.

3.2.3.1.3 Observation Component List

This section lists the types of components that can be sent at the start of an observation, the relative order in which they are dumped, their tags, and indicates where their structures can be located.

Order	Component	Tag	Structure Definition
1	Observation Header	OBS_COMP_HEADER	Section 3.2.2.1
2	Main DSP Load Block	OBS_COMP_MAIN_DSP	Section 3.2.2.1.1
3	Aux. DSP Load Block	OBS_COMP_AUX_DSP	Section 3.2.2.1.1
4	Partition Map	OBS_COMP_PARTMAP	**Partition Map Builder Module Specification**, Section 3.1
5	PCA Parameters	OBS_COMP_PCA	Section 3.2.2.2
6	Encoded Typer Table	OBS_COMP_TYPER	**Typer Builder Module Specification**, Section 3.2
7	Recoder Record	OBS_COMP_RECODER	Section 3.2.2.2.1
8	Recoder Corrections	OBS_COMP_REC_CORR	**Recoder Builder Module Specification**, Section 3.2.1
9	Recoder Groupings	OBS_COMP_REC_GROUP	**Recoder Builder Module Specification**, Section 3.2.2
10	Recoder Output Pos.	OBS_COMP_REC_POS	**Recoder Builder Module Specification**, Section 3.2.3
11	Binned Mode Param.	OBS_COMP_BINNED	**Binned Mode Module Specification**, Section 2.1
12	Burst Trigger Param.	OBS_COMP_TRIGGER	**Binned Mode Module Specification**, Section 2.1.1
11	Pulsar Fold Param.	OBS_COMP_PULSAR	**Pulsar Fold Mode Module Specification**, Section 2.1
11	Event Encoded Param	OBS_COMP_EVENT	**Event Encoded Mode Module Specification**, Section 2.1
11	Single Bit Mode Param.	OBS_COMP_SINGLEBIT	**Single Bit Mode Module Specification**, Section 2.1
11	Burst Catcher Param.	OBS_COMP_CATCHER	**Burst Catcher Mode Module Specification**, Section 2.1
11	FFT Mode Parameters	OBS_COMP_FFT	**FFT Mode Module Specification**, Section 2.1
5	Position Hist. Param.	OBS_COMP_POSITION	**Position Histogram Mode Module Specification**, Section 2.1
5	Mult. Timeseries Param.	OBS_COMP_TIMESER	**Multiple Timeseries Mode Module Specification**, Section 2.1
5	Event by Event Param.	OBS_COMP_ASMEVENT	**Event by Event Mode Module Specification**, Section 2.1

Table 1 Observation Parameter Dump Components

3.2.3.2 Science Data

All science data, taken during an observation, consists of a series of partitions. Each partition is sent to the ground using one or more telemetry packets. The first packet of a partition is called a Partition Header Packet, and has the format described in Section 3.2.3.2.1. These packets are assigned a "*Data Stream Id*" of STREAM_ID_PARTHDR, which is defined in Section 3.3.2.3. Subsequent packets of a partition are called Partition Data Packets, and have the format described in Section 3.2.3.2.2. These packets are assigned a "*Data Stream Id*" of STREAM_ID_PARTDATA, which is defined in Section 3.3.2.4. Please refer to the **Flight Software Global Design Specification**, Section 3.2.2 for a description of the telemetry packet headers and "*Data Stream Id*".

3.2.3.2.1 Partition Header Packet

Configuration Id (4 bytes) .... This field is identical to the Configuration Id in the Parameter Block used by the observation (see Section 3.2.2).

Observation Start Time (4 bytes)...... This specifies the actual (as opposed to scheduled) starting time of the observation (in MajorTicks).

[x,x,x,x,x,x,s,f] .... Partition Flags Bit-field (1 byte)

(x).............................. Unused

(s) .... Science telemetry has been suspended. No Partition Data packets will be sent for the partition. The "Words in Partition" field will be set to zero.

(f) .... Input FIFO spillage. If clear, DSP's Input FIFO did not fill during the acquisition of the partition's data. If set, then the FIFO overflowed. If a partition did not accumulate any data, but this flag is set, then the FIFO filled before the end of the previous partition.

Mode Specific Flags (1 byte)............ This byte is reserved for use by the specific science modes. Each mode is free to independently define the meaning of this field.

Partition Number (4 bytes) .... This identifies the partition within the observation, starting from zero and incrementing by one for each partition produced. Dropped partitions are indicated by gaps in the Partition Number.

Words in Partition (2 bytes)............. This specifies the total number of data words contained in the unpacked partition. These data words are packed and sent in subsequent Partition Data Packets. If set to zero, this value indicates that telemetry of the science data has been suspended and no partition data will be sent. Since the Partition Map can be configured to ignore an entire partition of data, it is possible to get a non-zero value in this field and yet receive no science data telemetry packets.

3.2.3.2.2 Partition Data Packet Format

Partition Sequence # (1 byte) .... This identifies the data in a sequence of partition data packets. Starting from zero, this value increments by one for each Partition Data Packet sent in a partition.
Reserved (1 byte) .... This byte is reserved to maintain word alignment.
Partition Data (N bytes) .... The rest of the packet contains the partition data.

3.2.3.3 Observation Coda Packet Format

At the end of an observation, the Science Thread checks the parameters and table values used during the observation against the parameters requested by the Observation Parameter Block. It then sends the results of these tests, and certain other observation information, to the ground using an Observation Coda Packet. Observation Coda telemetry packets are assigned a "*Data Stream Id*" of STREAM_ID_OBS_CODA, which is defined in Section 3.3.2.5. Please refer to the **Flight Software Global Design Specification**, "Packet Formats : Telemetry Packets" for a description of the telemetry packet headers and "*Data Stream Id*". The Observation Coda Packet data has the following format:

Configuration Id (4 bytes) .... This field is identical to the Configuration Id in the Parameter Block used by the observation (see Section 3.2.2).
Observation Start Time (4 bytes)...... This specifies the actual (as opposed to scheduled) starting time of the observation (in MajorTicks).
Partition Count (4 bytes)................... This specifies how many partitions of data were generated (but not necessarily telemetered) by the observation.
Typer Error Count (2 bytes) .... This is the total number of discrepancies between the encoded Typer Table and the actual Typer Table, after the observation (Value has meaning for PCA Modes Only).
Typer Table Offset (2 bytes) .... This identifies the Typer Table offset which contained the first discovered discrepancy (Value has meaning for PCA Modes Only).
Typer Table Value (2 bytes) .... This identifies the value that was read at the Typer Table offset, which didn't match the encoded table (Value has meaning for PCA Modes Only).
Recoder Error Count (2 bytes).......... This is the total number of discrepancies between the encoded Recoder Tables and the actual Recoder Table, after the observation (Value has meaning for PCA Modes Only).
Recoder Table Offset (2 bytes)......... This identifies the Recoder Table offset which contained the first discovered discrepancy (Value has meaning for PCA Modes Only).
Recoder Table Value (2 bytes) .... This identifies the value that was read at the Recoder Table offset, which didn't match the encoded table (Value has meaning for PCA Modes Only).

[x, x, x, x, x, x, x, p] .... Observation Error Flags (1 byte)
(x) .... Unused
(p) .... Partition Map Error. This bit indicates whether or not the Partition Map was corrupted during the observation. If set to 0, then the Partition Map was intact at the end of the observation. If set to 1, then the Partition Map was corrupted.

Mode Specific Error Flags (1 byte) .. This byte is reserved for use by the specific science modes. Each mode is free to independently define the meaning of this field.

3.3 Constants

3.3.1 Thread Identifiers

3.3.1.1 THREAD_ID_SCIENCE

This value uniquely identifies the Science Thread on a given EA. This value is used for the following:

a. Used by the Thread Manager Module to identify the thread (see **Thread Manager Module Specification**).

b. As the "*Destination Thread Id*" in Science Command Packets (see Section 4).

c. Provided in all Command Acknowledgment packets produced by the Science Thread (see Section 4).

3.3.1.2 THREAD_ID_HOUSEKEEPER

This is the Thread Id of the ASM Housekeeper Thread (see **Housekeeper Thread Module Specification**). Position Histogram Mode notifies this thread whenever the ASM interface becomes available for use.

3.3.2 Stream Data Size and Identifiers

3.3.2.1 FS_STREAM_SIZE

This indicates the number of bytes available in a telemetry packet for stream specific data. This constant is used by the Science Thread to determine how much science data can be packed into a telemetry packet (minus any science data packet header overhead).

3.3.2.2 STREAM_ID_OBS_PARAM

This value is stored in the "*Data Stream Id*" of a telemetry packet to indicate that the packet is part of an observation parameter dump (see Section 3.2.3.1).

3.3.2.3 STREAM_ID_PARTHDR

This value is stored in the "*Data Stream Id*" of a telemetry packet to indicate that the packet contains a partition header from an observation (see Section 3.2.3.2).

3.3.2.4 STREAM_ID_PARTDATA

This value is stored in the "*Data Stream Id*" of a telemetry packet to indicate that the packet contains partition data from an observation (see Section 3.2.3.2).

3.3.2.5 STREAM_ID_OBS_CODA

This value is stored in the "*Data Stream Id*" of a telemetry packet to indicate that the packet contains post-observation information (see Section 3.2.3.3).

3.3.3 Science Mode Limits and Identifiers

3.3.3.1 SCIENCE_MAX_MODES (255)

This defines the maximum number of science modes supported by the Science Thread. The Science Mode Identifier in the Observation Parameter Block (see Section 3.2.2.1) must be less than this value (since it is not negative, it can never be less than zero)..

3.3.3.2 PCA_MODE_BINNED

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Binned Mode.

3.3.3.3 PCA_MODE_PULSAR

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Pulsar Fold Mode.

3.3.3.4 PCA_MODE_EVENT

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Event Encoded Mode.

3.3.3.5 PCA_MODE_SINGLEBIT

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Single-Bit Code Mode.

3.3.3.6 PCA_MODE_CATCHER

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Burst Catcher Mode.

3.3.3.7 PCA_MODE_FFT

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in FFT Mode.

3.3.3.8 ASM_MODE_POSITION

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Position Histogram Mode.

3.3.3.9 ASM_MODE_MULTIPLE

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Multiple Time Series Mode.

3.3.3.10 ASM_MODE_EVENT

This value is stored in the Observation Parameter Block Header (see Section 3.2.2.1) and tells the Science Thread to run the observation in Event by Event Mode.

3.3.4 Observation Component Tags

3.3.4.1 SCIENCE_MAX_COMP (255)

This value identifies the maximum number of Observation Component tags supported by the Science Thread (i.e. the maximum supported by an unsigned 8-bit field minus 1, see 3.2.3.1.1).

3.3.4.2 OBS_COMP_HEADER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an Observation Parameter Block Header (see Section 3.2.2.1).

3.3.4.3 OBS_COMP_MAIN_DSP

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of the Main DSP Load Block (see Section 3.2.2.1.1), referenced by the Observation Parameter Block Header (see Section 3.2.2.1).

3.3.4.4 OBS_COMP_AUX_DSP

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of the Auxiliary DSP Load Block (see Section 3.2.2.1.1).

3.3.4.5 OBS_COMP_PARTMAP

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Partition Map (see **Partition Map Builder Module Specification**, Section 3.1).

3.3.4.6 OBS_COMP_PCA

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of PCA Parameter Block (see Section 3.2.2.2).

3.3.4.7 OBS_COMP_TYPER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an Encoded Typer Table (see **Typer Builder Module Specification**, Section 3.2).

3.3.4.8 OBS_COMP_RECODER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an Recoder Record (see Section 3.2.2.2.1).

3.3.4.9 OBS_COMP_REC_CORR

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Recoder Pulse Height Correction Table (see **Recoder Builder Module Specification**, Section 3.2.1)

3.3.4.10 OBS_COMP_REC_GROUP

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Recoder Pulse Height Grouping Table (see **Recoder Builder Module Specification**, Section 3.2.2)

3.3.4.11 OBS_COMP_REC_POS

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Recoder Pulse Height Output Position Table (see **Recoder Builder Module Specification**, Section 3.2.3)

3.3.4.12 OBS_COMP_BINNED

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Binned Mode specific parameter block (see **Binned Mode Module Specification**, Section 2.1).

3.3.4.13 OBS_COMP_TRIGGER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Binned Mode’s Burst Trigger Parameter Block (see **Binned Mode Module Specification**, Section 2.1.1).

3.3.4.14 OBS_COMP_PULSAR

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Pulsar Fold Mode specific parameter block (see **Pulsar Fold Mode Module Specification**, Section 2.1).

3.3.4.15 OBS_COMP_EVENT

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an Event Encoded Mode specific parameter block (see **Event Encoded Mode Module Specification**, Section 2.1).

3.3.4.16 OBS_COMP_SINGLEBIT

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Single Bit Mode specific parameter block (see **Single Bit Mode Module Specification**, Section 2.1).

3.3.4.17 OBS_COMP_CATCHER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Burst Catcher Mode specific parameter block (see **Burst Catcher Mode Module Specification**, Section 2.1).

3.3.4.18 OBS_COMP_FFT

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an FFT Mode specific parameter block (see **FFT Mode Module Specification**, Section 2.1).

3.3.4.19 OBS_COMP_POSITION

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Position Histogram Mode specific parameter block (see **Position Histogram Mode Module Specification**, Section 2.1).

3.3.4.20 OBS_COMP_TIMESER

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of a Multiple Time Series Mode specific parameter block (see **Multiple Time Series Mode Module Specification**, Section 2.1).

3.3.4.21 OBS_COMP_ASMEVENT

This value is stored in the Component Tag Field of a Component Header or Data packet (see Sections 3.2.3.1.1 and 3.2.3.1.2) to indicate that the packet contains a copy of an Event by Event Mode specific parameter block (see **Event by Event Mode Module Specification**, Section 2.1).

3.3.5 Science Thread Command Codes

3.3.5.1 CMD_SCIENCE_SUSPEND

This value is stored in the Command Opcode field of CCSDS command packets sent to the Science Thread (see **Flight Software Global Design Specification**, Section 3.2.1). It tells the Science Thread to stop sending science telemetry.

3.3.5.2 CMD_SCIENCE_RESUME

This value is stored in the Command Opcode field of CCSDS command packets sent to the Science Thread (see **Flight Software Global Design Specification**, Section 3.2.1). It tells the Science Thread to start sending science telemetry.

3.3.5.3 CMD_SCIENCE_TERMINATE

This value is stored in the Command Opcode field of CCSDS command packets sent to the Science Thread (see **Flight Software Global Design Specification**, Section 3.2.1). It tells the Science Thread to terminate the observation.

3.3.5.4 CMD_SCIENCE_AUTORESTART

This value is stored in the Command Opcode field of CCSDS command packets sent to the Science Thread (see **Flight Software Global Design Specification**, Section 3.2.1). It tells the Science Thread to restart the EA once the current observation has terminated and sent its coda.

3.3.6 Command Reply Codes

3.3.6.1 CMD_REPLY_OK

This reply code is sent with a command acknowledgment when a Science Command has been interpreted successfully (see Section 4). NOTE: The effects of some commands may take place after the acknowlegment has been set.

3.3.6.2 CMD_REPLY_NONE

This reply code is sent with a command acknowledgment when a Science Command had no effect due to the current state of the system.

3.3.6.3 CMD_REPLY_UNKNOWN

This reply code is sent with a command acknowledgement when a Science Command contained an operation code which was not recognized by the Science Thread.

3.3.6.4 CMD_REPLY_ALREADY

This reply code is sent with a command acknowledgement when a Science Command asks for an operation which has already been performed.

3.3.7 Memory Map Addresses

3.3.7.1 MMAP_SCIENCE_BUFFER_0 (0x030000 flat-address)

This is the address of the lower half of the Science Thread's partition buffer, which resides in the second-to-top 32K of the 80286's RAM. Whenever data is taken from one of the DSP's partition's, the lower 32K bytes are copied into this buffer. The remainder, if any, is copied into MMAP_SCIENCE_BUFFER_1 (see Section 3.3.7.2). Once a partition has been copied, the Science Thread analyzes, packs and telemeters the buffered data.

NOTE: The Science Buffer is broken into two pieces so that the Flight Software can avoid having to process exactly 1 segment's worth of data, and hence, avoid nasty segment boundary conditions.

3.3.7.2 MMAP_SCIENCE_BUFFER_1 (0x038000 flat-address)

This is the address of the upper half of the Science Thread's partition buffer, which resides in the top 32K of the 80286's RAM. Whenever data are taken from the DSP's partition which exceed 32K bytes, the lower 32K is copied to the lower 32K of the science buffer (see MMAP_SCIENCE_BUFFER_0 Section 3.3.7.1), and the remainder is copied to this location.

3.3.7.3 MMAP_SCIENCE_BUF_SIZE (0x08000 bytes)

This is the size of each of the two the Science Thread partition buffers (see Sections 3.3.7.1 and 3.3.7.2).

3.3.7.4 MMAP_DSP_FLAT_START (0x060000 flat-address)

This is the starting address of the DSP's external RAM, mapped into the 80286's address space.

3.3.7.5 MMAP_DSP_FLAT_END (0x07fffe flat-address)

This is the last valid address of the DSP's external RAM, mapped into the 80286's address space.

3.3.7.6 MMAP_DSP_BLOCK_SIZE (0x008000 bytes)

This is the size of any given DSP shared memory block (see Sections 3.3.7.7, 3.3.7.8, 3.3.7.9, 3.3.7.10). The DSP shared memory is segmented into 32K byte sections so that the 82086 never needs to perform operations which span the 64Kbyte segment size limit.

3.3.7.7 MMAP_DSP_SCRATCH_CODE (0x060000 flat-address)

This is the starting address of the DSP's external code RAM, mapped into the 80286's address space. The DSP Load blocks, which load code into the DSP, must set the first location of this area to the code's first instruction (see **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.11).

3.3.7.8 MMAP_DSP_SCRATCH_DATA (0x068000 flat-address)

This is the starting address of the DSP’s external scratch data RAM, mapped into the 80286’s address space (see **XTE EDS Hardware Functional Description and Requirements**, Section 2.1.4.11).

3.3.7.9 MMAP_DSP_PARTITION_0 (0x070000 flat-address)

This is the address of the start of the lower half of the DSP’s external histogram buffer RAM, in the 80286’s address space. The Science Thread copies science data paritions from this, and the MMAP_DSP_PARTITION_1 (see Section 3.3.7.10), regions of memory (see **XTE EDS Hardware Functional Description and Requirements**, Table 1).

3.3.7.10 MMAP_DSP_PARTITION_1 (0x078000 flat-address)

This is the address of the start of the upper half of the DSP’s external histogram buffer RAM, in the 80286’s address space. The Science Thread copies science data paritions from this, and the MMAP_DSP_PARTITION_0 (see Section 3.3.7.9), regions of memory (see **XTE EDS Hardware Functional Description and Requirements**, Table 1).

3.3.8 Science Thread Reset/Panic Message Codes

3.3.8.1 MSG_SCIENCE_AUTORESTART

This message code is passed to UTILpanic() when the Science Thread automatically restarts the EA at the end of an observation (see **CMD_SCIENCE_AUTORESTART** Section 3.3.5.4). Since it is caused by a CMD_SCIENCE_AUTORESTART command, this panic message does not indicate an error. Currently, this is the only message code not intended to indicate an error. The message should read something like:

"PANIC: ---science_reset() --Observation Ended. Restarting EA."

3.3.8.2 MSG_SCIENCE_DISPATCH_INVALID_MODE_ID

This message code is passed to UTILpanic() if science_dispatch() is called with an Observation Parameter Block which contains a invalid Science Mode identifier. The message should read something like:

"PANIC: ---science_dispatch(%d) --Handler Science Mode Identifier"

where %d is the Science Mode Id contained in the Observation Parameter Block.

3.3.8.3 MSG_SCIENCE_DISPATCH_UNKNOWN_MODE

This message code is passed to UTILpanic() if science_dispatch() is called with an Observation Parameter Block which contains a Science Mode identifier which does not correspond to any of the supported Science Modes. The message should read something like:

"PANIC: ---science_dispatch(%d) --Unknown Science Mode"

where %d is the Science Mode Id contained in the Observation Parameter Block.

3.3.8.4 MSG_SCIENCE_SETUP_OBS_NO_DSPBLK

This message code is passed to UTILpanic() if SCIENCEsetup_obshdr() is called with an Observation Parameter Block which does not specify any DSP code or data to load. The message should read something like:

"PANIC: ---SCIENCEsetup_obs() --No Main or Auxiliary DSP Load Block Specified"

3.3.8.5 MSG_SCIENCE_SETUP_PCA_NULL_REC

This message code is passed to UTILpanic() if SCIENCEsetup_pca() is called with an PCA Parameter Block which contains a NULL pointer to the Recoder Record (see Section 3.2.2.2.1). The message should read something like:

"PANIC: ---SCIENCEsetup_pca() --Recoder Record Pointer is NULL"

3.3.9 Common Science Mode Panic Message/Housekeeping Codes

3.3.9.1 MSG_SCIMODE_IV_DEAD_DSP

This message code is defined as a service to the various science modes. Whether or not this condition is detected is up to each specific mode. This message code is passed to UTILpanic() if a science mode's interval interrupt client function detects that the DSP is not processing events (usually by monitoring the expected First Flag vs. actual First Flag interrupts). The message should read something like:

"PANIC: ---Science Mode Interval Interrupt Client() --Crashed or Locked DSP"

3.3.9.2 MSG_SCIMODE_PROCESS_UNLOCKED_BUF

This message code is defined as a service to the various science modes. Whether or not this condition is detected is up to each specific mode. This message code is passed to UTILpanic() if a science mode's data processing routine discovers that the Science Buffer was not properly locked by the mode's First Flag interrupt client function. The message should read something like:

"PANIC: ---Science Mode Process() --Science Buffer Unlocked"

3.3.9.3 MSG_SCIMODE_PROCESS_BAD_PARTNUM

This message code is defined as a service to the various science modes. Whether or not this condition is detected is up to each specific mode. This message code is passed to UTILpanic() if a science mode's data processing routine discovers that partition number, as set by the mode's First Flag interrupt client function, is invalid. The message should read something like:

"PANIC: ---Science Mode Process() --Bad Partition Number"

3.3.9.3.1 LOG_SCIMODE_FF_DROP_PART

This Software Housekeeping Statistic is incremented whenever a given Science Mode's First Flag Interrupt Client function drops a partition of data due to a backlogged thread. Refer to the individual Science Mode Module Specifications (see Section 1.2) for a description of which specific function within the mode issues this statistic, and under what mode specific conditions.

3.3.10 Science Thread Bit-Field Definitions and Macros

3.3.10.1 SCIENCE_OBSHDR_PRIORITY(hdr) Macro

This macro, when passed a pointer to the Observation Parameter Block Header (see Section 3.2.2.1) supplies the science telemetry priority value stored in the "observation flags" field. This macro assumes that the priority field is 2-bits wide, starting at bit location 2.

3.3.10.2 SCIENCE_OBSHDR_GOVERNED(hdr) Macro

This macro, when passed a pointer to the Observation Parameter Block Header (see Section 3.2.2.1) supplies the state of the "governed" bit stored in the "observation flags" field. This macro assumes that the governed field is 1-bit wide, starting at bit location 1. This macro returns a logical 1 if the bit is set to 1, and a logical 0 if the bit is set to 0.

3.3.10.3 SCIENCE_OBSHDR_SUSPENDED(hdr) Macro

This macro, when passed a pointer to the Observation Parameter Block Header (see Section 3.2.2.1) supplies the state of the "suspended" bit stored in the "observation flags" field. This macro assumes that the suspended field is 1-bit wide, starting at bit location 0. This macro returns a logical 1 if the bit is set to 1, and a logical 0 if the bit is set to 0.

3.3.10.4 SCIENCE_PARTHDR_SPILLED (0x001)

This mask selects the "spillage" bit within the Partition Header Packet’s "Partition Flags" field (see Section 3.2.3.2.1).

3.3.10.5 SCIENCE_PARTHDR_SUSPENDED (0x002)

This mask selects the "suspended" bit within the Partition Header Packet’s "Partition Flags" field (see Section 3.2.3.2.1).

3.3.10.6 SCIENCE_CODA_PARTMAP_ERR (0x001)

This mask selects the "Partition Map Error" flag within the "Observation Error Flags" field within the Observation Coda Packet (see Section 3.2.3.3).

3.3.11 Science Thread Resource Constants

3.3.11.1 SCIENCE_MAX_PARTBYTES (FS_STREAM_SIZE - 2)

This defines the maximum number of bytes of science data that can be telemetered in a single Science Data Telemetry Packet (see Section 3.2.3.2).

3.3.11.2 SCIENCE_MAX_PARTWORDS (SCIENCE_MAX_PARTBYTES/2)

This defines the maximum number of 16-bit words of science data that can be telemetered in a single Science Data Telemetry Packet (see Sections 3.3.11.1 and 3.2.3.2).

3.3.11.3 SCIENCE_STACK_SIZE (2048)

This defines the number of words in the Science Thread’s stack space.

3.3.11.4 SCIENCE_POOL_SIZE (8256)

This defines the number of bytes in the Science Thread’s telemetry buffer pool.

3.3.12 Science Thread Internal Enumerated Types

3.3.12.1 SCIENCE_STATE

This set defines the various states of the Science Thread. The following lists and describes each state:

3.3.12.1.1 SCIENCE_STATE_SETUP

The Science Thread starts in this state, and remains in this state until data acquisition has started. Except for the CMD_SCIENCE_AUTORESTART command, commands received while in this state have no effect and generate a CMD_REPLY_NONE acknowledgment (see Section 3.3.5).

3.3.12.1.2 SCIENCE_STATE_RESUMED

The Science Thread enters this state when data acquisition is in progress and the Science Thread is not suspending its science data packet telemetry. The Science Thread accepts all commands while in this state, although CMD_SCIENCE_RESUME will cause a CMD_REPLY_ALREADY acknowledgment (see Sections 3.3.5, and 3.3.6).

3.3.12.1.3 SCIENCE_STATE_SUSPENDED

The Science Thread enters this state when data acquisition is in progress and the Science Thread is suspending its science data packet telemetry. The Science Thread accepts all commands while in this state, although CMD_SCIENCE_SUSPEND will cause a CMD_REPLY_ALREADY acknowledgment (see Sections 3.3.5, and 3.3.6).

3.3.12.1.4 SCIENCE_STATE_TERMINATED

The Science Thread enters this state after receiving a CMD_SCIENCE_TERMINATE command, or once it has determined that an observation has ended. The Science Thread acts on only CMD_SCIENCE_AUTORESTART commands while in this state. All other commands cause either a CMD_REPLY_NONE or CMD_REPLY_ALREADY acknowledgement (see Sections 3.3.5, and 3.3.6).

3.3.12.2 SCIENCE_SEND

This set of codes are used as return values from SCIENCEsend_partdata(). The following lists and describes each of these codes:

3.3.12.2.1 SCIENCE_SEND_DONE

The current partition has been completely sent. Call SCIENCEnew_partition() before processing the next partition.

3.3.12.2.2 SCIENCE_SEND_MORE

A telemetry packet has been packed and posted, but there is more data to pack in the partition. Call SCIENCEsend_partdata() again to process more of the partition.

3.3.12.2.3 SCIENCE_SEND_NOBUF

SCIENCEsend_partdata() failed to obtain a telemetry buffer (i.e. they're all in the telemetry queue waiting to be transferred). Call SCIENCEsend_partdata() again to retry.

4 Commands

4.1 Command and Acknowledgment Packet Formats

4.1.1 Command Packets

The Science Thread obtains command packets using facilities provided by the UPLINK Module (see **Uplink Manager Module Specification**). The format of command packets are described in Section 3.2.1. The command specific descriptions assume that the "*Destination Thread*" portion of the command "*EDS Command Byte 0*" is set to THREAD_ID_SCIENCE (see Section 3.3.1.1). Each of the following command descriptions only specify the "*Command Opcode*" of the "*EDS Command Byte 0*" and the format of any additional data stored in the "*EDS Optional Command Bytes 2-N*." Please refer to the **Flight Software Global Design Specification**, Section 3.2.1, for a description of the overall format of command packets, and of the "*EDS Command Bytes*".

If the Science Thread receives a command packet with a "*Command Opcode*" which it doesn't understand, it will send a reply with the following reply code, and ignore the command:

CMD_REPLY_UNKNOWN .... Command Opcode is unknown to the Science Thread. This reply does not supply any reply specific data.

4.1.2 Acknowledgment Packets

The Science Thread sends a Command Acknowledgment Packet to the ground for each command packet received, using facilities provided by the UTIL module. The format of Command Acknowledgment Packets is described in the **Utility Module Specification**, Section 3.1.3. Each Command Acknowledgment packet contains a copy of the entire command packet being acknowledged (see Section above), an Acknowledgment Reply Code, and command specific Reply Data. Each of the command descriptions below, describes the various Acknowledgment Reply Codes and data that can be expected as a result of each command.

4.2 Science Suspend Command

Suspend telemetry from an Observation.

Format

Thread Id = THREAD_ID_SCIENCE
Opcode = CMD_SCIENCE_SUSPEND
Arguments = None.

Description

When the Science Thread receives this command, it will reply to the command, and then suspend telemetry from the current observation. When an observation suspends, it finishes processing and transmitting the current partition (i.e. the partition already copied from the DSP, as opposed to the partition currently being collected by the DSP). It then continues to acquire and process data, but it doesn't attempt to downlink it. This command has no arguments.

Acknowledgment Codes

CMD_REPLY_NONE .... Observation not running. Suspend nop's.
This reply does not supply any reply specific data.
CMD_REPLY_ALREADY .... Observation is already suspened. Extra Suspend nop's.
This reply does not supply any reply specific data.
CMD_REPLY_OK........................... Suspend Accepted. Observation marked to suspend.
This reply does not supply any reply specific data.

Error Conditions and Handling

The following scenarios are possible and are handled as follows:

No observation is currently active (ie. suspended or running)
Emit CMD_REPLY_NONE
IGNORE Suspend
The current observation is already suspended
Emit CMD_REPLY_ALREADY
IGNORE Suspend

4.3 Science Resume Command

Resume telemetry from a suspended Observation.

Format

Thread Id = THREAD_ID_SCIENCE
Opcode = CMD_SCIENCE_RESUME
Arguments = None.

Description

When the Science Thread receives this command, it replies to the command, finishes acquiring the current partition and then resumes science data telemetry from the current observation. This command has no arguments.

Acknowledgment Codes

CMD_REPLY_NONE .... Observation not running. Resume nop's.
This reply does not supply any reply specific data.
CMD_REPLY_ALREADY .... Observation is already sending telemetry. Resume nop's.
This reply does not supply any reply specific data.
CMD_REPLY_OK........................... Resume Accepted. Observation marked to resume.
This reply does not supply any reply specific data.

Error Conditions and Handling

The following scenarios are possible and are handled as follows:

No observation is currently active (ie. suspended or running)
Emit CMD_REPLY_NONE
IGNORE Resume
The current observation is already running and sending telemetry (ie. not suspended)
Emit CMD_REPLY_ALREADY
IGNORE Resume

4.4 Science Terminate Command

Cleanly Terminate an Observation.

Format

Thread Id = THREAD_ID_SCIENCE
Opcode = CMD_SCIENCE_TERMINATE
Arguments = None.

Description

When the Science Thread receives this command, it replies to the command, finishes the current partition being telemetered, and then terminates the observation. Once the observation has terminated, the Science Thread generates and telemeters the observation coda. This command takes no arguments.

Acknowledgment Codes

CMD_REPLY_ALREADY .... Observation has already been terminated. No effect.
This reply does not supply any reply specific data.
CMD_REPLY_NONE .... Observation not running. Nothing terminated
This reply does not supply any reply specific data.
CMD_REPLY_OK .... Terminate Accepted. Observation will end cleanly.
This reply does not supply any reply specific data.

Error Conditions and Handling

The following scenarios are possible and are handled as follows:

Terminate command has already been received
Emit CMD_REPLY_ALREADY
IGNORE second Terminate
Observation has already self terminated (i.e. no configuration or end of dwell sequence)
Emit CMD_REPLY_ALREADY
IGNORE Terminate Command
Observation data acquisition (run stage) has not started
Emit CMD_REPLY_NONE
IGNORE Terminate

4.5 Science AutoRestart Command

Restart EA after End of Observation

Format

Thread Id = THREAD_ID_SCIENCE
Opcode = CMD_SCIENCE_AUTORESTART
Arguments = None.

Description

When the Science Thread receives this command, it replies to the command, and sets an internal reset-upon-completion flag. If this flag is set when the observation ends, the Science Thread will wait for all telemetry activity to cease, and then reset the EA, via a panic message. This command takes no arguments.

Acknowledgment Codes

CMD_REPLY_ALREADY .... Restart Flag Already Asserted. No operation. This reply does not supply any reply specific data.
CMD_REPLY_OK........................... AutoRestart Command Succeeded. Restart flag asserted. This reply does not supply any reply specific data.

Error Conditions and Handling

The following scenario is possible and is handled as follows:

AutoRestart command already received
Emit CMD_REPLY_ALREADY
IGNORE second AutoRestart command

5 Interrupts

5.1 Interval Interrupt (Level - 1)

The Science Thread uses the **Timing Module** to install Interval Interrupt client functions. These functions, once installed, are invoked by the **Timing Module** while handling Interval Interrupts caused by Interval Boundaries.

5.2 First-Flag Interrupt (Level - 6)

The Science Thread uses the **Timing Module** to install First-Flag Interrupt client functions. These functions, once installed, are invoked by the **Timing Module** while handling First-Flag Interrupts, which are caused by the DSP reading Interval Markers from its Input FIFO when First-Flag interrupt generation is enabled.

6 Events

6.1 EV_NEW_UPLINK

The **UPLINK Module** notifies the Science Thread of this event whenever a new uplink command packet is available and is addressed to the Science Thread.

6.2 EV_SCI_BUF_AVAIL

The **Buffer Pool Utility Module** notifies the Science Thread with this event whenever a Science telemetry packet has been transferred out of the EA, and the telemetry buffer has been released to an empty Science Telemetry Pool.

6.3 EV_MAJOR

The **Timing Module** notifies all threads with this event whenever a Major-tick pulse occurs. The Science Thread sometimes waits for this event when governing the science telemetry rate.

6.4 EV_SCI_DATA_RDY

The First-Flag interrupt client function, installed by the Science Thread, notifies the Science Thread with this event after it has copied a partition of science data from the DSP's RAM to the Science Data Buffer. The Science Thread uses this event to determine when to start analyzing, packing, and telemetering a new partition.

6.5 EV_NO_MORE_DATA

The First-Flag interrupt client function, installed by the Science Thread, notifies the Science Thread when it has determined that no more data is to be produced for a given observation.

6.6 EV_ASM_IF_AVAIL

The Science Thread, when running Position Histogram Mode, notifies the **ASM Housekeeper Thread** whenever the ASM Command/Housekeeping Interface is available for use by the **ASM Housekeeper Thread**. The **ASM Housekeeper Thread** uses this event to determine when to accumulate hardware housekeeping information from the SSC's and Drive Assembly. This ensures that rotation commands and housekeeping commands are not sent along the ASM interface at the same time. Please refer to the **Housekeeper Thread Module Specification** for a description of the operation of the **ASM Housekeeper Thread**.

7 Functions

7.1 SCIENCEinit() - Initializes Science Thread Context

Usage

void SCIENCEinit(void)

Description

SCIENCEinit() initializes the context of the Science Thread, and is intended to be called once by the STARTUP module during system initialization. SCIENCEinit() uses the **Thread Manager Module** to construct the thread's descriptor, and add the descriptor to the system's list of active threads.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must be called once, during system initialization.

2) This function must be called after the **Thread Manager Module** has been initialized.

See Also

Thread Manager Module Specification, **Startup Module Specification**.

7.2 Science Mode Function Prototype

Usage

void <*MODE*>run_mode(UINT8 **hdr*);

Description

Each Science Mode module consists of a function <*MODE*>run_mode(), where <*MODE*> is the mnemonic for the science mode module. For example, BINNEDrun_mode() would correspond to the Science Mode module which runs the Binned Science Mode. These functions accept a standard set of arguments. "*hdr*" points to the Observation Parameter Block Header (please refer to Section 3.2.2.1). The mode specific parameters are located immediately after the header. These functions do not return until the observation is terminated.

Science mode functions must perform the following operations:

1) Setup the EA's Front End, based on the Observation and Mode's parameters (see Section 3.2.2).

2) Send a complete observation parameter dump (see Section 3.2.3.1).

3) Setup and run the Science Mode and telemeter the science data (see Section 3.2.3.2).

4) Handle command packets sent to the Science Thread while the observation is running (see Section 4).

5) Check the tables and parameters at the end of an observation, and send an observation coda (see Section 3.2.3.3).

Return Values

NONE - FUNCTIONS ARE VOID

7.3 SCIENCEget_buffer() - Obtain Science Telemetry Buffer

Usage

UINT8 *SCIENCEget_buffer (SCIENCE_STATE *in_state*,
SCIENCE_STATE **out_state*);

Description

SCIENCEget_buffer() obtains a telemetry buffer from the Science Thread's telemetry buffer pool. "*in_state*" is the current state of the Science Thread, upon entry to this function. "*out_state*" is a pointer to where this function should store the "new" current state of the thread.

If one or more commands arrive while SCIENCEget_buffer() is waiting for a telemetry buffer to become available, SCIENCEget_buffer() will call SCIENCEhandle_uplink() for each received command, using the state returned by SCIENCEhandle_uplink() as the current state of the thread (i.e. if another command arrives, the returned state is then passed back to SCIENCEhandle_uplink()).

If *"out_state"* is not NULL, SCIENCEget_buffer() stores the final state value in the location pointed to by "*out_state*". If "*out_state*" is NULL, the final output state is not stored. This is useful when the thread is in a state which prevents commanded state changes (such as the setup, dump and idle phases of the observation).

Return Values

A pointer to the obtained telemetry buffer.

Caveats

1) This function will wait forever if a Science Thread telemetry buffer never becomes available.

2) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

SCIENCEhandle_uplink(), **Thread Manager Module Specification, Buffer Pool Module Specification**

7.4 SCIENCEhandle_uplink() - Get/Handle a Command Packet

Usage

SCIENCE_STATE SCIENCEhandle_uplink (SCIENCE_STATE *state*);

Description

SCIENCEhandle_uplink() obtains and handles any pending command packets addressed to the Science Thread and returns any state change warranted by the handled comand. "*state*" is the current state of the Science Thread upon entry to this function. If no command packets are pending for the Science Thread, SCIENCEhandle_uplink() returns immediately with "*state*" as the return value. If a command is pending, SCIENCEhandle_uplink() consumes the command, modifies any necessary state variables, generates the acknowledgment to the command, releases the command's buffer, and returns the new state warranted by the command.

Refer to Section 4 for descriptions of the supported command set. The following lists the thread states, commands and replies and new states generated by each:

"state" / Command	CMD_SCIENCE_RESUME	CMD_SCIENCE_SUSPEND	CMD_SCIENCE_TERMINATE
SCIENCE_STATE_SETUP	CMD_REPLY_NONE "*state*"	CMD_REPLY_NONE "*state*"	CMD_REPLY_NONE "*state*"
SCIENCE_STATE_RESUMED	CMD_REPLY_ALREADY "*state*"	CMD_REPLY_OK SCIENCE_STATE_SUSPENDED	CMD_REPLY_OK SCIENCE_STATE_TERMINATED
SCIENCE_STATE_SUSPENDED	CMD_REPLY_OK SCIENCE_STATE_RESUMED	CMD_REPLY_ALREADY "*state*"	CMD_REPLY_OK SCIENCE_STATE_TERMINATED
SCIENCE_STATE_TERMINATED	CMD_REPLY_NONE "*state*"	CMD_REPLY_NONE "*state*"	CMD_REPLY_ALREADY "*state*"

CMD_SCIENCE_AUTORESTART is not sensitive to the passed *"state"* variable. On the first receipt of CMD_SCIENCE_AUTORESTART, a CMD_REPLY_OK is generated. Subsequent CMD_SCIENCE_AUTORESTART commands cause a CMD_REPLY_ALREADY response and always returns the passed "*state*" value.

Unknown commands generate a CMD_REPLY_UNKNOWN response and return "*state*" to the caller.

Return Values

"*state"* if no commands were received, or the next state as described in the table above.

Caveats

1) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

Uplink Module Specification, **Utility Module Specification**, **Buffer Pool Module Specification**.

7.5 SCIENCEsetup_obshdr() - Setup using Observation Block

Usage

void SCIENCEsetup_obshdr (SCIENCE_OBS_HDR **hdr*);

Description

SCIENCEsetup_obshdr() configures elements of the Flight Software and Hardware based on parameters provided in the Main Observation Parameter Block (see Section 3.2.2.1), pointed to by "*hdr*". This function is responsible for configuring the following items, using the specified methods:

Item	Header Parameter Field	Action
Bin Duration/Nibble Mask	TimeBin Duration	None. Mode is responsible for configuring the TimeBin duration and Timestamp Nilbble mask, typically using TIMINGtime_bin().
Partition Map	Partition Map Pointer	The Partition Map Pointer is passed to PARTMAPsetup() to configure the map.
DSP Memory	Main DSP Load Block Auxiliary DSP Load Block	Initially fills shared DSP Memory with zeros and then copies the code/data from the Main and Auxiliary DSP Load Blocks, in that order. Null Main or Auxiliary DSP Load block pointers are ignored (i.e. the correspondiing block is not loaded). If both DSP Load Block pointers are NULL, this function issues a MSG_SCIENCE_SETUP_OBS_NO_DSPBLK panic message and resets the EA.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function assumes that the DSP is held in a reset state.

2) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

Timing Module Specification, Partition Map Module Specification

7.6 SCIENCEsetup_pca() - Setup using PCA Parameter Block

Usage

void SCIENCEsetup_pca (SCIENCE_OBS_HDR **hdr*);

Description

SCIENCEsetup_pca() configures elements of the Flight Software and Hardware based on parameters provided in the PCA Parameter Block (see Section 3.2.2.2), which abuts the Main Observation Parameter Block, pointed to by "*hdr",* in all PCA-EA specific Science Modes. This function is responsible for configuring the following items, using the specified methods:

Item	Header Parameter Field	Action
Bin Duration/Nibble Mask	Nibble Mask	None. Mode is responsible for configuring the TimeBin duration and Timestamp Nilbble mask, typically using TIMINGtime_bin().
Typer Lookup Table	Typer Table Pointer	The Typer Table Pointer is passed to TYPERbuild() to setup the lookup table.
Recoder Lookup Table	Recoder Record	If the Recoder Record pointer is not NULL, it sets up the Recoder Lookup table using RECODERbuild(). If the pointer is NULL, it uses a MSG_SCIENCE_SETUP_PCA_NULL_REC panic message and resets the EA, using UTILpanic().

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function assumes that the DSP is held in a reset state.

2) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

Timing Module Specification, Typer Builder Module Specification, Recoder Builder Module Specification

7.7 SCIENCEdump_obs() - Dump Observation Parameter Block

Usage

void SCIENCEdump_obs (UINT8 *pri*, SCIENCE_OBS_HDR **hdr*);

Description

SCIENCEdump_obs() dumps the contents of the Main Observation Parameter Block (see Section 3.2.2.1), pointed to by *"hdr*", and all blocks referenced by the parameter block. It uses the telemetry priority channel specified by "*pri*". If "*hdr"* is NULL, nothing is dumped. If any reference pointer is NULL, SCIENCEdump_obs() skips the corresponding component. The components are dumped in the following order:

Order	Name	Component Tag	Length Determination
1	Observation Parameter Block	OBS_COMP_HEADER	sizeof()
2	Main DSP Load Block	OBS_COMP_MAIN_DSP	internal computation
3	Auxiliary DSP Load Block	OBS_COMP_AUX_DSP	internal computation
4	Partition Map	OBS_COMP_PARTMAP	PARTMAPlength()

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

Partition Map Module Specification, Downlink Module Specification

7.8 SCIENCEdump_pca() - Dump PCA Parameter Block

Usage

void SCIENCEdump_pca (UINT8 *pri*, SCIENCE_OBS_HDR **hdr*);

Description

SCIENCEdump_pca() dumps the contents of the PCA Parameter Block (see Section 3.2.2.2) and all blocks referenced by the PCA Parameter Block. The PCA Parameter Block abuts the Main Observation Parameter Block, pointed to by "*hdr",* in all PCA-EA specific Science Modes. SCIENCEdump_pca() uses the telemetry priority channel specified by "*pri*". If "*hdr*" is NULL, nothing is dumped. If any reference pointer is NULL, SCIENCEdump_obs() skips the corresponding component. The components are dumped in the following order:

Order	Name	Component Tag	Length Determination
1	PCA Parameter Block	OBS_COMP_PCA	sizeof()
2	Typer Parameters	OBS_COMP_TYPER	TYPERencoded_length()
3	Recoder Record	OBS_COMP_RECODER	sizeof()
4	Recoder Correction Parameters	OBS_COMP_REC_CORR	RECODERencoded_lengths()
5	Recoder Grouping Parameters	OBS_COMP_REC_GROUP	RECODERencoded_lengths()
6	Recoder Output Position Parameters	OBS_COMP_REC_POS	RECODERencoded_lengths()

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

Typer Builder Module Specification, Recoder Builder Module Specification, Downlink Module Specification

7.9 SCIENCEdump_blk() - Dump a Parameter Block

Usage

void SCIENCEdump_blk (UINT8 *pri*, UINT32 *config*,
UINT8 *tag*, UINT8 **addr*, UINT16 *length*);

Description

SCIENCEdump_blk() dumps a contiguous region of memory as a part of the Observation Parameter Set. "*pri*" selects the telemetry priority to use, "*config*" is the Configuration Id, as stored in the Observation Parameter Block Header (see Section 3.2.2.1). "*tag*" identifies the component being dumped (see Section 3.2.3.1.3). "*addr*" points to the block of data to dump, and "*length*" is the number of bytes to dump. Refer to Section 3.2.3.1.1 and 3.2.3.1.2 for descriptions of the Component Header Packet Format and Component Data Packet Format. If "*addr*" is NULL, or "*length*" is zero, the component will not be dumped.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function must only be called by functions running as part of the Science Thread (science modes for example).

2) This function ensures that the command queue is kept drained by calling SCIENCEhandle_uplink() once for each telemetry packet posted, passing SCIENCE_STATE_SETUP as the thread state and ignoring any returned state changes (since we're in a setup state, no state changes should occur). It also uses SCIENCEget_buffer() when obtaining telemetry buffers.

See Also

SCIENCEget_buffer(), SCIENCEhandle_uplink(), **Downlink Module Specification**

7.10 SCIENCEnew_partition() - Start Partition of Science Data

Usage

void SCIENCEnew_partition (UINT16 **addr*, UINT16 *length*);

Description

SCIENCEnew_partition() causes the Science Thread to start processing a new partition of science data. "*addr*" points to the starting location of the partition data, and "*length*" is the number of 16-bit words in the partition.

Return Values

NONE - FUNCTION IS VOID

Caveats

1) This function MUST be called once before processing each partition.

2) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

SCIENCEsend_partdata(), **Partition Map Module Specification**

7.11 SCIENCEsend_partdata() - Pack/Post Science Data Packet

Usage

SCIENCE_SEND SCIENCEsend_partdata (UINT8 *priority*);

Description

SCIENCEsend_partdata() attempts to pack and send one telemetry packet's worth of partition data from the current partition specified by SCIENCEnew_partition(). "*priority*" specifies the telemetry priority to use.

Return Values

SCIENCE_SEND_DONE	-Partition has been completely sent. Call SCIENCEnew_partition() to start a new partition.
SCIENCE_SEND_MORE	-Packet sent. Call again to process more data in partition.
SCIENCE_SEND_NOBUF	-Packet not sent. No telemetry buffer was available.

Caveats

1) SCIENCEnew_partition() must be called before processing a new partition of data.

2) This function must only be called by functions running as part of the Science Thread (science modes for example).

See Also

SCIENCEnew_partition(), **Partition Map Module Specification, Downlink Module Specification**

8 Detailed Design Considerations

8.1 Data Flow

The following illustrates the overall data flow within the Science Thread Module:

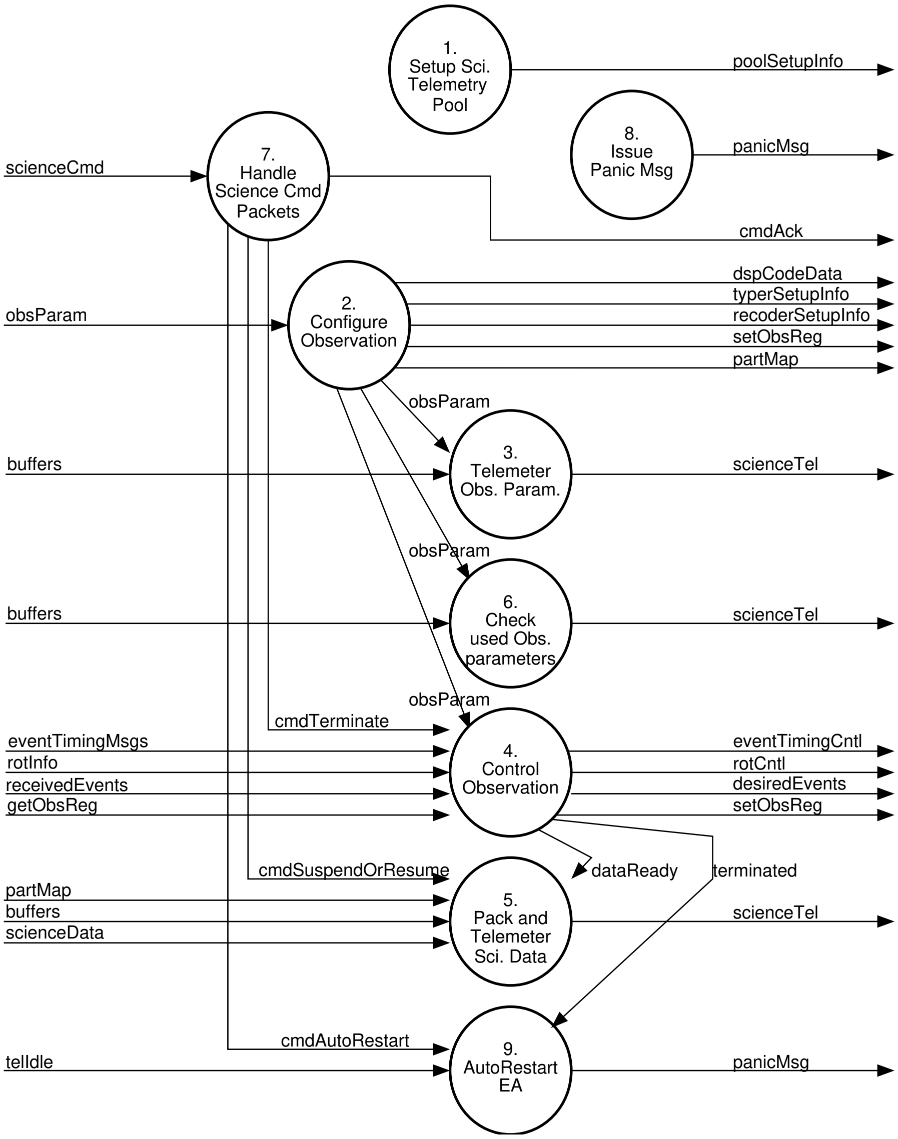

Figure 9 Science Thread Module Data Flow

When the running Science Thread first gets control (bubble 1), it initializes its state variables, and sets up a telemetry buffer pool (poolSetupInfo). All Observation Parameter Dump Packets, Partition Header and Data Packets, and Observation Coda Packets (collectively indicated as scienceTel) use telemetry packet buffers from this pool (buffers).

The Science Thread then obtains its Observation Parameter Block (obsParam), and uses the block to configure the observation (bubble 2). The mode id in the configuration block determines which mode function to call, which then configures the hardware and any state variables which it requires.

Once the mode function has configured the system, it telemeters (bubble 3) the parameters used by the observation to the ground. Once all of the parameters have been telemetered, the mode function schedules and starts the observation (bubble 4). As the observation runs, science data is accumulated, analyzed, and packed into telemetry packets, which are then sent to the ground (bubble 5).

During the observation, the mode function periodically checks for uplinked science commands (bubble 7). If the command tells the Science Thread to terminate (cmdTerminate) the observation, the mode function finishes sending the current data set, and ends the observation. At the end of the observation, the mode function checks any lookup tables and parameters used during the observation, and sends the result of the checks in the Observation Coda (bubble 6). The mode function also handles requests to suspend or resume science telemetry (cmdSuspendorResume).

If, during the life of the Science Thread, a command is sent to restart the EA upon termination of the current observation (cmdAutoRestart), the request is recorded in an internal state variable (bubble 9). Once the observation terminates (terminated), the Science Thread waits for telemetry activity to cease (telIdle). Assuming that the Spacecraft is dedicated to taking telemetry from only this EA, and the EA is only producing science telemetry, this will take under 1 second. Once all of the telemetry from the observation has been transferred out of the EA, the Science Thread issues a panic message and resets the EA (panicMsg).

During the life of the Science Thread, various integrity checks are made (bubble 8), to ensure that the system has not been corrupted. If an error is encountered, the Science Thread will emit a panic message (panicMsg) to the UTIL module, which will then post the message and reset the Event Analyzer. Upon reboot, the Event Analyzer is reinitialized, and the Science Thread is restarted. Since the original parameters are stored in either ROM or Patch RAM, the Science Thread will then restart the observation.

8.2 Control Considerations

8.2.1 Thread Control Durations

Since the Flight Software uses a non-preemptive multitasking system, it is each thread's responsiblity to relinquish control often enough to ensure that other threads are run in a timely fashion (refer to the **Thread Manager Module Specification**, Section 6.2.2). The frequency at which the Science Thread needs to get control is dependent on the rate at which it needs to analyze, pack, and telemeter science data, and the maximum expected command rate to the Science Thread.

1) Uplink Command Rate:
The Spacecraft Data System (SDS) ensures that there is at least 30ms between command packets (**XTE-EDS Hardware Functional Description and Requirements**, Section 2.1.4.8.1). In practice, however, the ground software controls the actual command delivery rate, by specifying packet delivery times to the SDS. Since the Science Thread does not need to accept a series of closely spaced commands (autorestart, suspend, resume and terminate commands are typically separated by minutes, not milliseconds), the uplink rate has little impact on the Science Thread's control frequency requirements. If, for some reason, the scheduled command rate exceeds the capability of the Science Thread to process the command packets, the UPLINK module (see **Uplink Manager Module Specification**) will discard command packets, recording the number of discarded packets in software housekeeping. In this scenario, the Science Thread continues to operate as if the command was never sent.

2) Data Processing and Telemetry Rate:
First, assume that a given Science Thread needs to produce 32 telemetry packets per second (~256K bits/second), and that the Science Thread takes ~16ms to analyze, pack and post a single telemetry packet. If the Science Thread yields control between each telemetry packet (once every ~32ms), on average it must regain control within ~16ms of the yield in order to maintain the 256Kbit/sec. telemetry rate. If the Science Thread yields control between every 32 telemetry packets (once every ~500ms), on average it must regain control within ~500ms of the yield. In practice, the frequency at which the Science Thread yields while producing telemetry, will be somewhere between once every 32ms and once every 500ms, and is governed by the number of packets which are buffered by the thread, and the rate at which the SDS receives these packets. To ensure that the worst case conditions are met, when the Science Thread is producing telemetry packets at a high rate, it must obtain control within 16ms after it yields.

The Science Thread shall relinquish control (Twait) at least twice per second (once every 500ms). The Science Thread, when producing telemetry at a high rate, must obtain control within 16ms after it has relinquished control. If this requirement is not met, the Science Thread will continue the observation, but partitions of science data may be dropped. Dropped partitions are detected using the "*Partition Number*" field of the partition headers (see Section 3.2.3.2.1).

8.2.2 Interval Interrupt and First-Flag Interrupt Client Functions

Each science mode function will install its own Interval Interrupt, and First-Flag Interrupt client functions, using the TIMING module (see **Timing Module Specification**). During Interval Interrupt processing, the installed Interval Interrupt client function is called by the TIMING module. During First-Flag Interrupt processing, the TIMING module calls the First-Flag Interrupt client function. Since these functions are called inside interrupt handling, the design of these functions must be aware of the following:

1) The execution time of the client function must be tailored to the interrupt priority and expected frequency.

2) Data elements which are modified by the client and polled by a thread are considered volatile. If the compiler has a feature which caches data elements in registers, this feature must be disabled for these data elements.

3) Interrupt client functions have restrictions on the types of functions they can call. The detailed design of each of the client functions must ensure that all functions called by the client function can be called at interrupt time.

8.3 Module Structure

8.3.1 Main Science Thread Module Structure

The following diagram illustrates the structure of the top level Science Thread. Bolded boxes and text represent functions provided by other modules. The dotted line from SCIENCEinit to science_thread illustrates that SCIENCEinit installs science_thread as the first function to call when the Thread Manager schedules the thread for the first time. The dashed box, <MODE>run_mode, illustrates that science_dispatch calls one of a set of mode specific routines, based on the mode id in the observation parameter block. The structure of the mode specific functions depend on the specific function, and are described in detail in the mode specific documents (see Section 1.2). All mode functions, however, do have some common structural properties, which are described in Section 8.3.2.

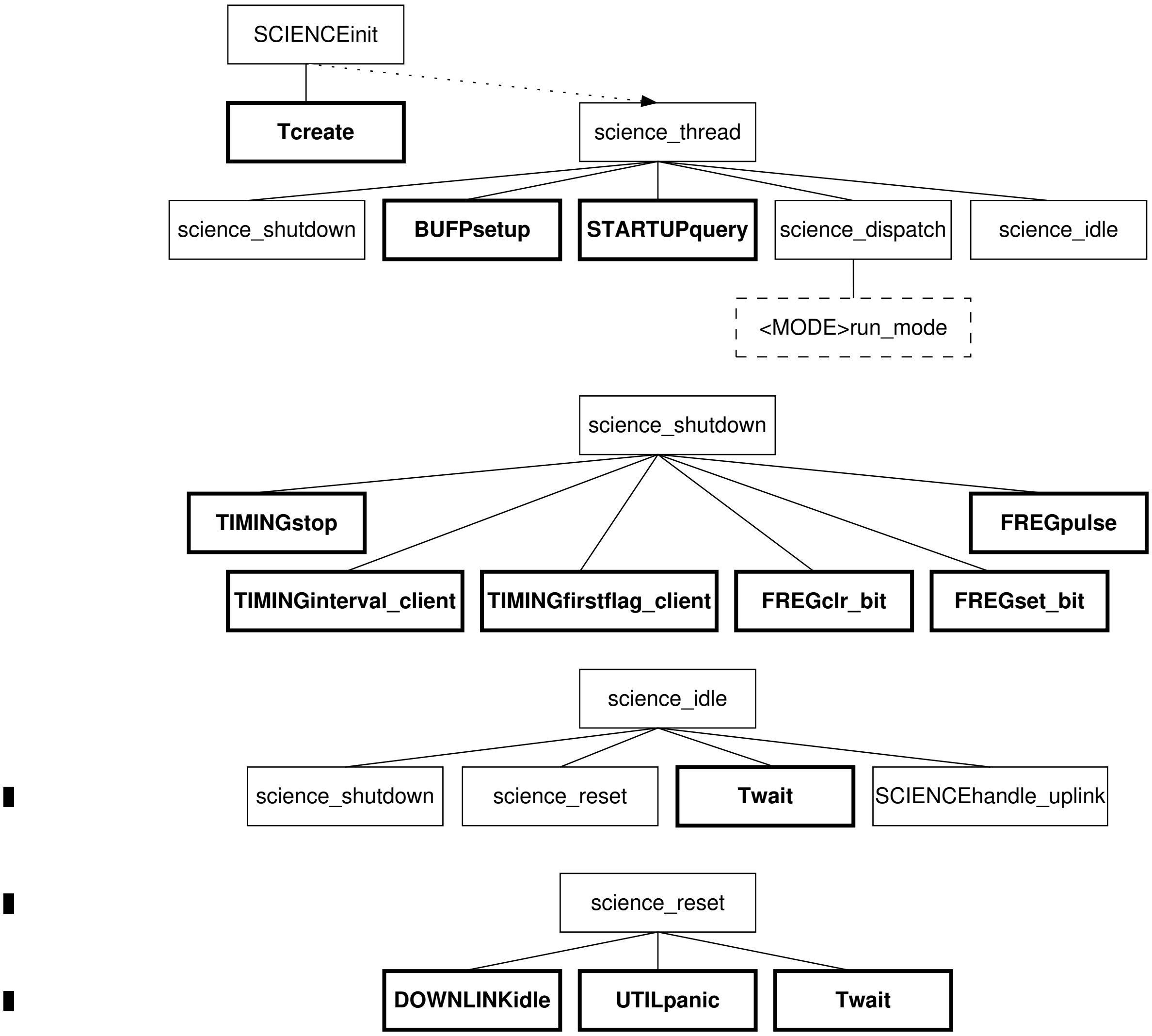

Figure 10 Main Science Thread Module Structure

Figure 10 does not show calls to the **Interrupt Manager** functions INTRbegin_atomic() and INTRend_atomic(), nor, with the exception of science_reset(), to the **Utility** function UTILpanic(). The following describes each of the functions shown in Figure 10. The details of each function's operation and arguments are described in the Science Thread Module's detailed design document.

SCIENCEinit .... This function is called by the **Startup Module** during system initialization. SCIENCEinit calls Tcreate in the **Thread Manager Module** to construct the context of the thread to be run. SCIENCEinit passes the thread's id (THREAD_ID_SCIENCE, Section 3.3.1.1), the thread's stack space and size, and the execution start address (science_thread) of the thread.

science_thread .... This function is invoked once by the **Thread Manager**, once multitasking has started. This function must never return. science_thread starts by ensuring that the system is in a known state by calling science_shutdown. science_thread then initializes the telemetry buffer pool using the **Buffer Pool Utility Module** (BUFPsetup). It then calls the **Startup Module** (STARTUPquery) to obtain a pointer to the EA's observation parameter block (format is described in Section 3.2.2). science_thread then runs the observation, by calling science_dispatch. Once the observation is complete, and science_dispatch returns, science_thread idles the thread by calling science_idle. science_idle never returns.

science_shutdown .... This function is called by science_thread and science_idle to ensure that the system is in a known state. science_shutdown calls the **Timing Module** (TIMINGstop, TIMINGinterval_client, and TIMINGfirst_flag_client) to ensure that all of the front end timing circuitry is turned off. science_shutdown also calls the **Front Registers Module** (FREGclr_bit, FREGpulse, and FREGset_bit) to ensure that the burst trigger output lines are in an inactive state, that the burst trigger input latches have been reset, and that the DSP is held in a reset state. NOTE: Interval generation is disabled while the DSP is held in a reset state.

science_dispatch .... This function is called by science_thread to run an observation. science_dispatch uses the science mode in the observation parameter block (see Section 3.2.2) to select one of the mode specific functions to run (<MODErun_mode). Once the function has been selected, science_dispatch calls the selected function. science_dispatch will panic the EA, using the **Utility Module** (UTILpanic), if the Science Mode Identifier in the parameter block is out of range

(MSG_SCIENCE_DISPATCH_INVALID_MODE_ID) or does not correspond to one of the supported Science Modes (MSG_SCIENCE_DISPATCH_UNKNOWN_MODE).

science_idle .... This function is called by science_thread to throw the Science Thread into an idle state. science_idle starts by calling science_shutdown to ensure that the front end hardware is in a safe state. It then enters a polling loop. If the internal reset-upon-termination flag is asserted, science_idle calls science_reset to complete telemetry activity and reset the EA. If the reset flag is not asserted, it then uses the **Thread Manager Module** (Twait) to wait for any pending uplink commands. If a command arrives, it calls SCIENCEhandle_uplink to respond to any command packets that may have been received. Once SCIENCEhandle_uplink returns, science_idle loops and rechecks the reset-upon-termination flag. This process of checking the flag, waiting for a packet, and handling the packet, repeats forever until a CMD_SCIENCE_AUTORESTART command is received (thus asserting the reset flag), or until the EA is reset explicity by the ground.

science_reset .... This function is called by science_idle to complete all telemetry activity from the observation and reset the EA. science_reset consists of a loop which ends when the EA is finally reset. Each iteration polls the **Downlink Module** (DOWNLINKidle). If telemetry is not yet idle, science_reset uses the **Thread Manager Module** (Twait) to yield control, allowing other threads to perform their tasks. Once the Science Thread regains control, it repeats the loop, retesting the telemetry activity. Once the **Downlink Module** indicates that all telemetry from the observation has been transferred out of the EA, science_reset uses the **Utility Module** (UTILpanic) to issue a message (MSG_SCIENCE_AUTORESTART) and reset the EA.

<MODE>run_mode .... These functions are called by science_thread to run an observation. An overview of the structure of these functions is described in Section 8.3.2. For detailed descriptions of these functions, refer to the mode specific module specifications, listed in Section 1.2.

8.3.2 Mode Function Structures

The following diagram illustrates some of the common structural properties of all mode specific functions, and some of the utility functions provided by the main Science Thread:

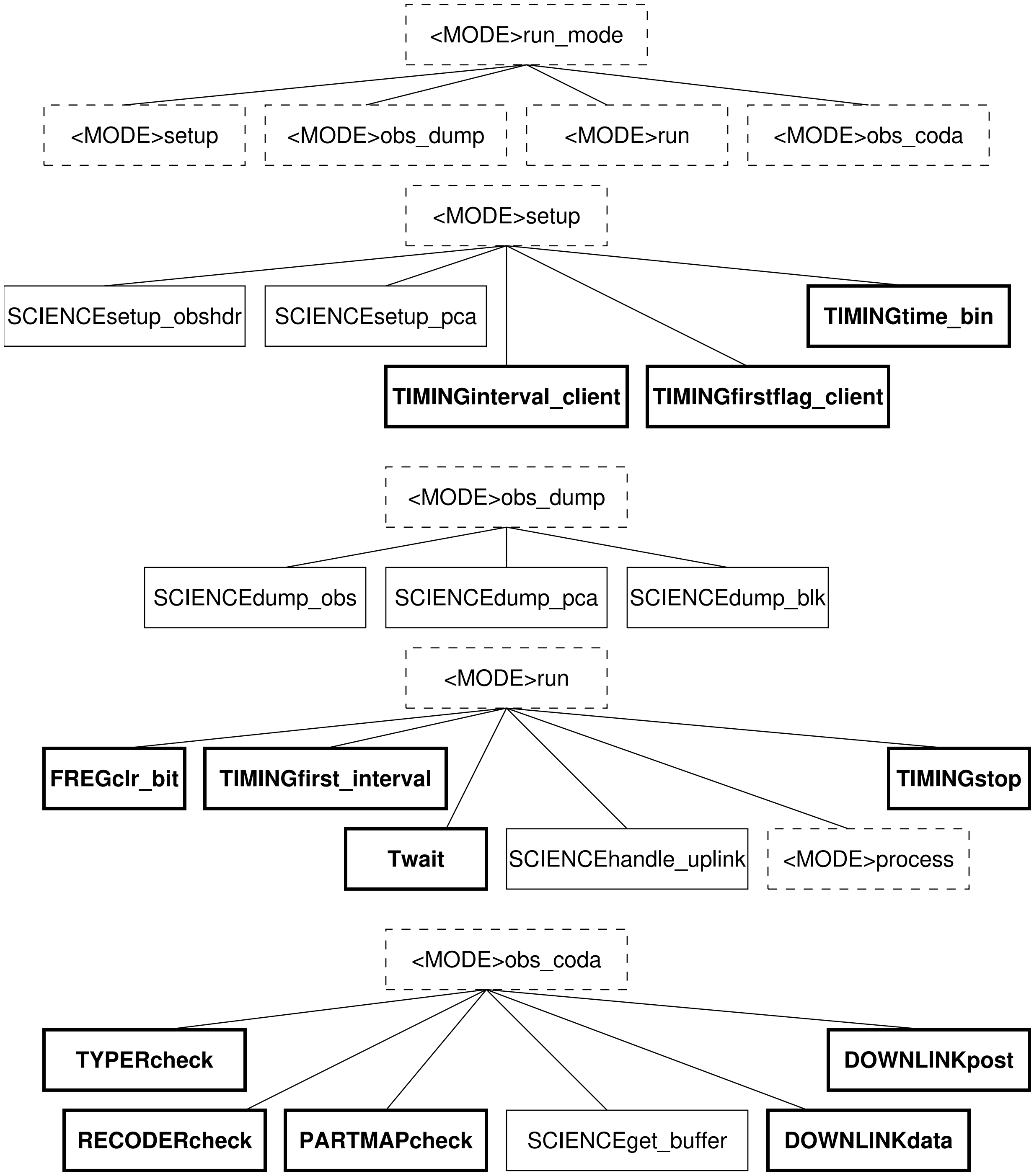

Figure 11 Mode Function Structures

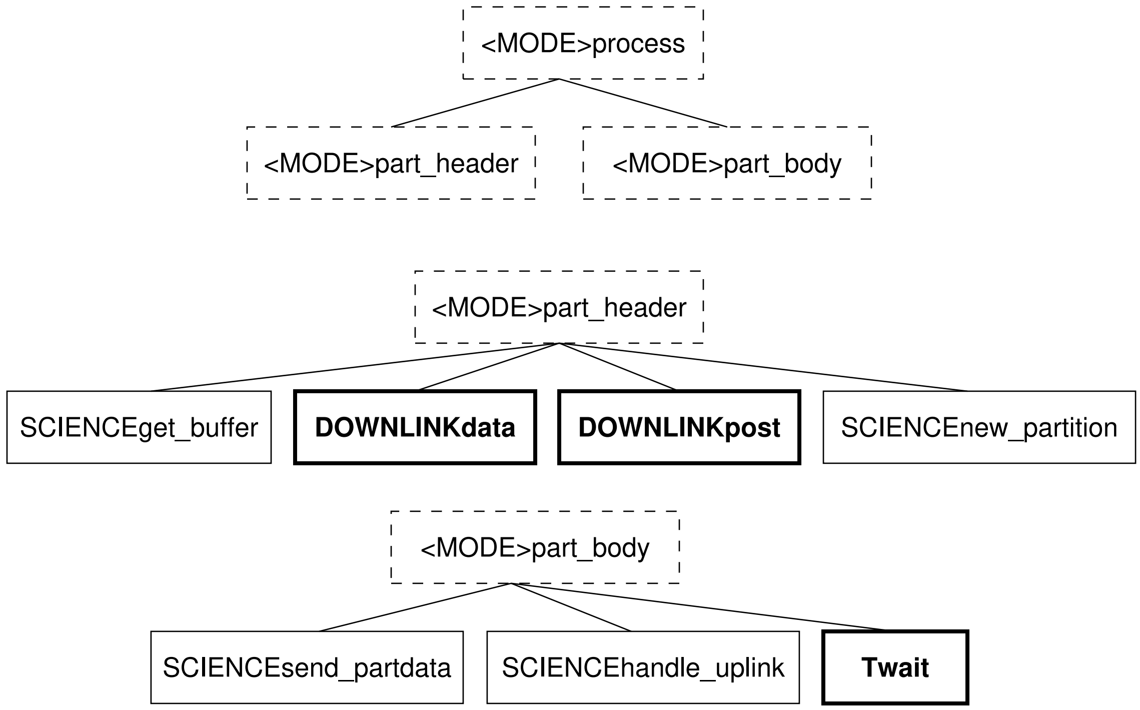

Figure 12 Mode Partition Processing Structures

Figures 11 and 12 do not show calls to the **Interrupt Manager** functions INTRbegin_atomic() and INTRend_atomic(), nor to the **Utility** function UTILpanic(). The following describes each of the functions shown in Figures 11 and 12. The details of each function's operation and arguments are described in the Science Thread Module's detailed design document, or in the science mode specific module and detailed design documents.

<MODE>run_mode .... These functions are called by science_dispatch to run a given science mode. These functions have a common structure. First, a given mode function must setup the front end hardware and any state variables (<MODE>setup). The function must then telemeter the parameters used for the observation (<MODE>obs_dump). Once the parameters are sent, the mode function can then start running the mode (<MODE>run). Once the observation completes, the mode functions must check the parameters used by the observation and send the results of the test in an observation coda (<MODE>obs_coda).

<MODE>setup .... These functions are called by their corresponding <MODE>run_mode functions to setup the hardware and any state variables for a given science mode.. These functions use parameters stored in the Observation Parameter Block. These functions may call some Science Thread utility functions to assist in this process. The Science Thread utility functions called may include SCIENCEsetup_obshdr and SCIENCEsetup_pca, which are described in Section 8.3.3. The mode specific setup functions may also call the **Timing Module** to install client functions (TIMINGinterval_client, TIMINGfirst_flag_client) and set the time-bin duration and mask (TIMINGtime_bin).

<MODE>obs_dump .... These functions are called by their corresponding <MODE>run_mode functions to dump the Observation Parameter Block parameters to the ground. These functions may call some Science Thread utility functions to assist in this process. The Science Thread utility functions called may include SCIENCEdump_obs, SCIENCEdump_pca, and SCIENCEdump_blk, which are described in Section 8.3.3.

<MODE>run.................................... These functions are called by their corresponding <MODE>run_mode functions to run the actual observation. These functions use the **Front Registers Module** (FREGclr_bit) to deassert the DSP's Reset Line, allowing the DSP to start execution. The mode specific run functions also call the **Timing Module** (TIMINGfirst_interval) to schedule and start an Interval sequence (since TIMINGfirst_interval always takes at least 1 second before starting to run, the modes do not have to explictly wait for the DSP to perform its initialization operations, assuming the DSP startup operations take under 1 second). These functions also use the **Thread Manager Module** (Twait) to wait for various events. If commands arrive while waiting, these functions may use the Science Thread utility function, SCIENCEhandle_uplink, to handle any received commands. Once a partition of data is acquired, these functions typically dispatch to their partition processing routine, <MODE>process.

<MODE>obs_coda .... These functions are called by their corresponding <MODE>run_mode functions to check the parameters used by the observation and telemeter the result to the ground. These functions call the **Partition Map Builder Module** (PARTMAPcheck) to check the integrity of the partition map, the **Typer Builder Module** (TYPERcheck) to check the integrity of the Front End's Typer Table, and

the **Recoder Builder Module** (RECODERcheck) to test the integrity of the Front End's Recoder Table. These functions may use the Science Thread utility function, SCIENCEget_buffer, to obtain a telemetry buffer for the coda, and the **Downlink Module** (DOWNLINKdata) to obtain a pointer to the start of usable data space in the telemetry buffer. Once the observation coda packet has been built, these functions call the **Downlink Module** (DOWNLINKpost) to post the packet to telemetry.

<MODE>process .... These functions are called by their corresponding <MODE>run functions to process a single partition of science data. These functions may check the integrity of any internal state variables, and issue panic messsages, using the **Utility Module** (UTILpanic, not shown). Some such panic messages may include MSG_SCIMODE_PROCESS_UNLOCKED_BUF, and MSG_SCIMODE_PROCESS_BAD_PARTNUM (see Section 3.3). They then call their respective partition header generation functions, <MODE>part_header, and partition data processing functions, <MODE>part_data.

<MODE>part_header .... These functions are called by their corresponding <MODE>process functions to generate and emit a Partition Header Telemetry Packet (see Section 3.2.3.2.1). They may call the utility function SCIENCEget_buffer, to obtain a science telemetry buffer. They then use the **Downlink Module** (DOWNLINKdata) to obtain a pointer to the application data area of the buffer. Once the buffer has been filled in, they use the **Downlink Module** (DOWNLINKpost) to post the packet for transfer out of the EA. They then use the Science Thread utility function, SCIENCEnew_partition, to reset the partition data packing state variables for a new partition of science data.

<MODE>part_body .... These functions are called by their corresponding <MODE>process functions to telemeter, and if appropriate, analyze a partition of science data. In most cases, these functions use SCIENCEsend_partdata to form and post a single Partition Data Telemetry Packet (see Section 3.2.3.2.2). In the process of sending the partition, these functions may use the utility function, SCIENCEhandle_uplink, to poll for and handle any pending uplink commands. They also use the **Thread Manager Module** (Twait) to wait for specific events, or to periodically yield control, allowing other threads to run.

8.3.3 Science Utility Function Structures

The following diagrams illustrate the structure of the utility functions provided by the Science Thread Module:

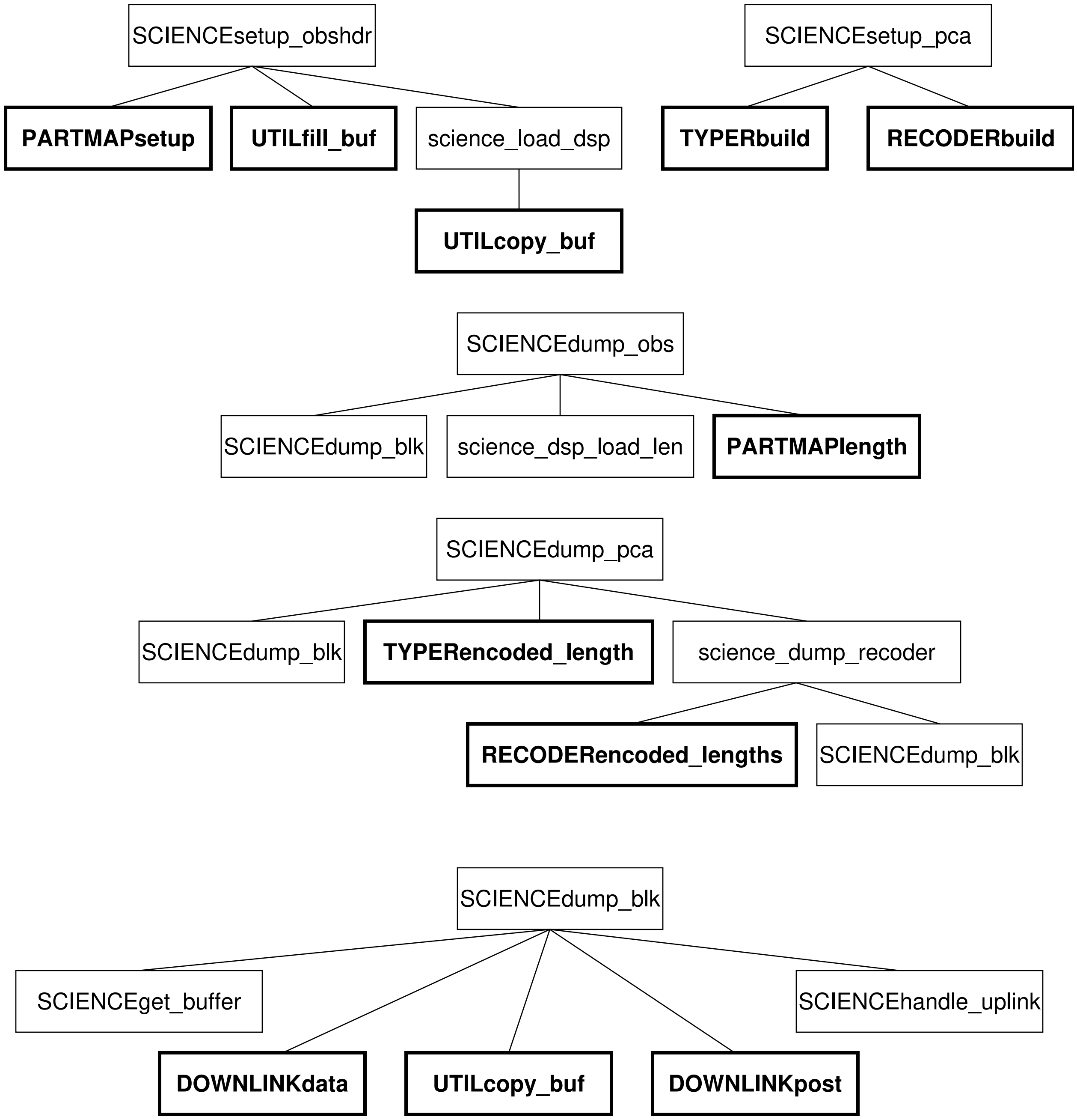

Figure 13 Science Observation Setup/Dump Utility Function Structures

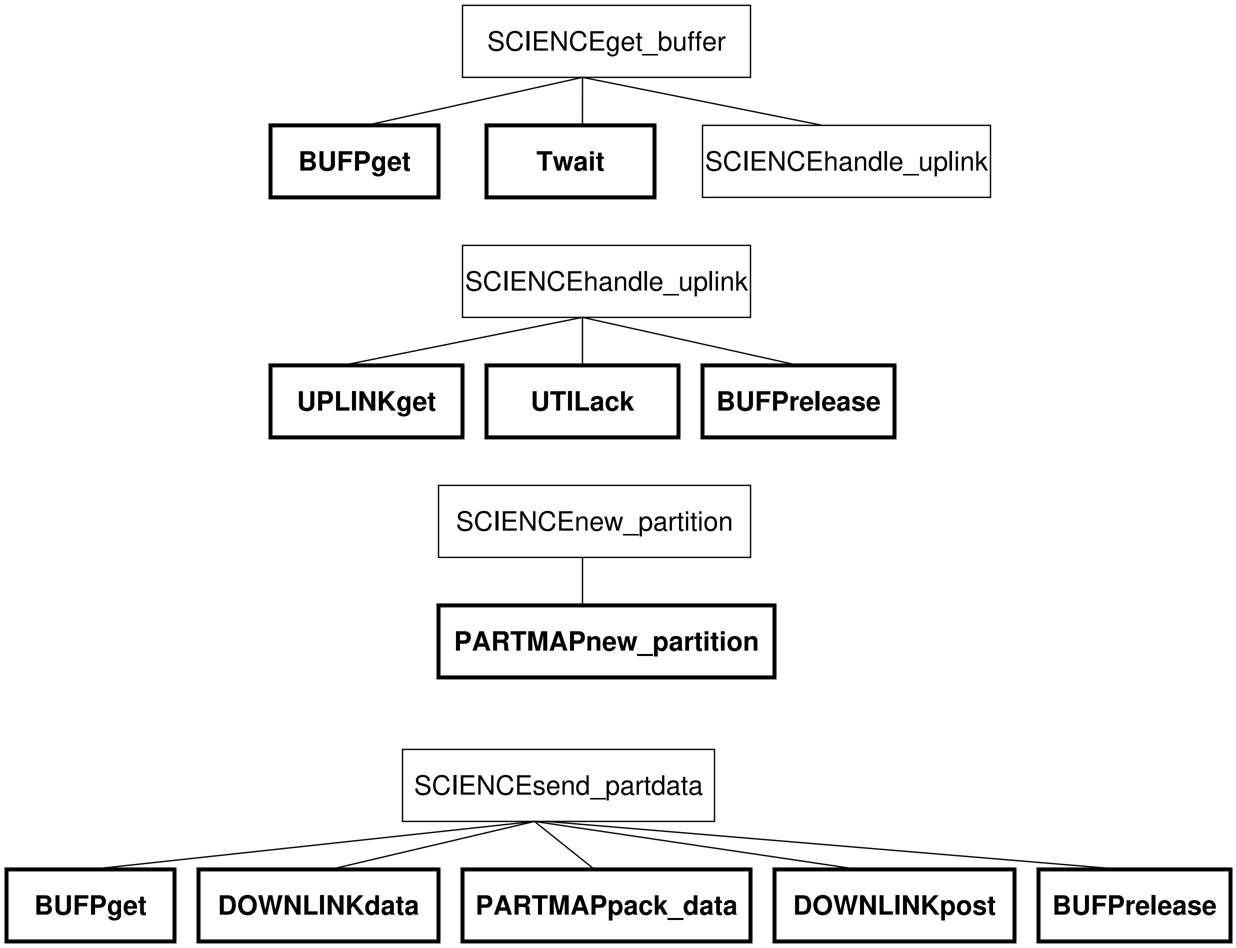

Figure 14 Science Miscellaneous Utility Function Structures

Figures 13 and 14 do not show calls to the **Interrupt Manager** functions INTRbegin_atomic() and INTRend_atomic(), nor to the **Utility** function UTILpanic(). The following describes each of the functions shown in Figures 13 and 14. The details of each function's operation and arguments are described in the Science Thread Module's detailed design document.

SCIENCEsetup_obshdr .... This function is called by the various mode specific setup routines (<MODE>setup). This function obtains setup information from the Observation Parameter Block Header (Section 3.2.2.1). It uses the **Partition Map Builder** (PARTMAPsetup) to load a partition map, the **Utility Module** (UTILfill_buf) to fill the DSP memory with zeros, and science_load_dsp to load DSP code and data into the DSP's external RAM. If this function is passed a configuration which specifies neither a Main DSP Load Block nor Auxiliary DSP Load Block (see Section 3.2.2.1), SCIENCEsetup_obshdr will panic the EA (MSG_SCIENCE_SETUP_OBS_NO_DSPBLK), using the **Utility Module** (UTILpanic, not shown).

SCIENCEsetup_pca .... This function is called by the various PCA mode specific setup routines (<MODE>setup). This function obtains PCA setup information from the PCA Parameter Block (Section 3.2.2.2). It uses the **Typer Builder Module** (TYPERbuild) to decode an encoded typer table into the PCA front end's Typer Lookup Table. It also uses the **Recoder Builder Module** (RECODERbuild) to decode the encoded recoder tables into the PCA front end's Recoder Lookup Table. If this function obtains a PCA Parameter Block (which immediately follows the passed Observation Parameter Block, see Sections 3.2.2.1 and 3.2.2.2) whose Recoder Record structure pointer is NULL, SCIENCEsetup_pca will panic the EA (MSG_SCIENCE_SETUP_PCA_NULL_REC) using the **Utility Module** (UTILpanic, not shown).

SCIENCEdump_obs .... This function is called by the various mode specific observation parameter dump routines (<MODE>obs_dump) to dump the elements contained in, and referenced by, the main Observation Parameter Block Header (Section 3.2.2.1). This function uses SCIENCEdump_blk to send the contents of the Observation Parameter Block Header, the Partition Map, if present, and the main and auxiliary dsp code/data load blocks, if present. It uses the **Partition Map Module** (PARTMAPlength) to obtain the length of the partition map, and calls science_dsp_load_len to separately compute the length of the Main and Auxiliary DSP Load Blocks.

SCIENCEdump_pca .... This function is called by the various PCA mode specific observation parameter dump routines (<MODE>obs_dump) to dump the elements contained in, and referenced by the PCA Parameter Block (Section 3.2.2.2). This function uses SCIENCEdump_blk to send the contents of the PCA Parameter Block, and the encoded typer table. It uses science_dump_recoder to dump the encoded recoder tables. SCIENCEdump_pca uses the **Typer Builder Module** (TYPERencoded_length) to obtain the length of the encoded typer table.

SCIENCEdump_blk .... This function is called by various Science Thread and mode specific functions to construct and telemeter science telemetry packets containing dumps of a contiguous piece of configuration data. This function uses the Science Thread utility function, SCIENCEget_buffer, to obtain science telemetry packets. As packets are obtained, this function uses the **Downlink Module** (DOWNLINKdata) to obtain a pointer to the data area of the packet buffer and fills in the parameter dump header or dump data information. It then uses the **Utility Module** (UTILcopy_buf) to copy the parameter information into the packet, and then uses the **Downlink Module** (DOWNLINKpost) to post the packet to telemetry. If, during this process, a command packet arrives, this function calls SCIENCEhandle_uplink to obtain, process, and acknowledge the command.

SCIENCEget_buffer .... This function is called by various functions within the Science Thread and the mode specific functions to wait for and obtain a science telemetry buffer. SCIENCEget_buffer consists of a loop. The loop terminates when a buffer is successfully obtained. It uses the **Buffer Pool Module** (BUFPget) to attempt to get a telemetry buffer from the science telemetry buffer pool. If a buffer is obtained, it returns with the pointer to the buffer. If not, it enters the body of the loop. The body uses the **Thread Manager Module** (Twait) to wait for a buffer to become available, or for an uplink command to arrive. If a command arrives, SCIENCEget_buffer calls SCIENCEhandle_uplink to process the command. Once a buffer is obtained, SCIENCEget_buffer provides the caller with a pointer to the obtained buffer and any state changes which occurred while waiting for the buffer.

SCIENCEhandle_uplink .... This function is called by various functions within the Science Thread and the mode specific functions. It attempts to obtain a command packet from the **Uplink Module** (UPLINKget). If a command packet is present, it executes the command based on the current state of the Science Thread. It then issues a command acknowledgment, using the **Utility Module** (UTILack), and releases the command packet's buffer using the **Buffer Pool Utility Module** (BUFPrelease). NOTE: UPLINKget() can be used to poll for uplink packets. A Twait() for an EV_NEW_UPLINK event is not required before calling UPLINKget().

SCIENCEnew_partition .... This function is called by the various mode specific functions responsible for running the observation (<MODE>run). This function resets some Science Thread Module data packing state variables, and sets up the **Partition Map Module** (PARTMAPnew_partition) for a new partition of data.

SCIENCEsend_partdata .... This function is called by the various mode specific functions responsible for running the observation (<MODE>run). This function packs and sends a single science telemetry packet, containing part of a partition of science data. This function obtains a science telemetry packet using the **Buffer Pool Utility Module** (BUFPget). It then uses the **Downlink Module** (DOWNLINKdata) to obtain a pointer to the data area of the packet buffer. It then fills the packet with packed science data. This function performs the packing operation using the **Partition Map Module** (PARTMAPpack_data). Once the packet has been filled, it posts the packet to be telemetered using the **Downlink Module** (DOWNLINKpost). If the **Partition Map Module** indicates that no data was packed into the buffer (i.e. the partition has been completely processed), SCIENCEsend_partdata releases the obtained buffer using the **Buffer Pool Module** (BUFPrelease).

science_load_dsp .... This function is called twice by SCIENCEsetup_obshdr. Once to load the main DSP load block into the DSP's external RAM, and a second time to load the auxiliary DSP load block into the DSP's external RAM. This function copies the DSP's code or data using the **Utility Module** (UTILcopy_buf).

science_dsp_load_len .... This function is called by science_dump_dsp to calculate the length, in bytes, of a DSP Load Block.

science_dump_recoder .... This function is called by SCIENCEdump_pca to send copies of the Recoder Record (Section 3.2.2.2.1), and the Recoder's Correction, Grouping, and Output Tables (see **Recoder Builder Module Specification**, Section 3.2). This function uses the **Recoder Builder Module** (RECODERencoded_lengths) to obtain the lengths of each of the encoded tables, and then uses SCIENCEdump_blk to build and send the telemetry packets containing the dumped data.

9 Requirements Traceability

9.1 Science Requirements

The Science Thread Module addresses many PCA and ASM Science Requirements. The requirements which have been addressed by this document are listed in the **Requirements Traceablity Matrix**. Most of the remaining PCA or ASM Science Requirements are addressed by the Science Mode module specifications.

10 Testing Considerations

10.1 Unit Testing

To perform unit tests on this module, the following equipment is required:

a. An Event Analyzer equipped with the Intermetrics ROM Monitor Debug Board

b. A workstation running the Intermetrics debugger

c. An RS-232 serial cable between the ROM Monitor's serial port and the workstation

d. A logic analyzer

To unit test this module, the tester must link and locate this module's functions to some EA RAM memory location. Using a resident monitor/debugger, the tester should first verify the operation of all functions internal to this module, and then verify the interactions between all functions internal to the module. These tests should verify all branches, calls, and legal and illegal parameters. The tester should then verify all calls made to functions outside of this module, and ensure that the passed arguments are always within the limits specified by the corresponding module specifications. The tester should also verify that the caller always handles values returned by these functions in an appropriate manner.

In addition to these tests, the tester should perform the following:

a. Verify that all functions which interpret components of the Observation Parameter Block extract the proper items and test the items for valid ranges. These functions include: science_dispatch, SCIENCEsetup_obshdr, SCIENCEsetup_pca, science_load_dsp, science_dsp_load_len, SCIENCEdump_obs, SCIENCEdump_pca, and science_dump_recoder.

b. Verify each set of mode specific functions, in conjunction with the main Science Thread.

10.2 Integration Testing

For integration tests, the following equipment is required:

a. An Event Analyzer

b. A populated System Manager board

c. A logic analyzer

d. Hardware which can:

   1. Drive the MinorTick and MajorTick clock signals (S/C Simulator)

   2. Receive telemetry packets from the EA (S/C Simulator).

   3. Block telemetry packets from the EA (i.e. stop receiving packets).

   4. Send simulated PCA events to the EA (PCA-Simulator).

   5. Send simulated ASM events to the EA (ASM-Simulator).

The integration tests should verify the operation of the Science Thread Module in conjunction with the other FS modules. These tests should include:

a. Verify that the Science Thread properly gets control after a reboot.

b. Verify that the Science Thread properly obtains the EA specific Observation Parameter Block from the STARTUP module.

c. Verify that the Science Thread properly configures the front end hardware before the start of the observation, given a variety of configuration settings.

d. Verify that the Science Thread accurately telemeters the Observation Parameters before the start of the observation.

e. Measure the execution times of each function, given all control paths through the function. This information, along with mode specific measurements, are used to calculate maximum execution times, which are then used to ensure that the thread's temporal requirements can be met (see Section 8.2.1). Since the bulk of the Science Thread's operation involves packing and posting science data, pay special attention to the time it takes for SCIENCEsend_partdata to pack and post a telemetry packet, given a variety of Partition Maps.

f. Verify that the Science Thread properly performs the desired mode, using the configured parameters. These tests should verify the operation of the mode under the following conditions:

   1. Ideal conditions (i.e. expected event rates, "safe" parameter settings, low science telemetry rates, etc.)

2. Telemetry overthroughput conditions. Ensure that data is lost as discrete partitions, and that the science telemetry stream properly indicates the loss.

3. Event overthroughput conditions. Ensure that if the DSP's Input FIFO overflows, the loss is indicated properly in the Partition Header corresponding to the time at which the data was lost.

4. Suspend and Resume functionality. Ensure that, when the Science Thread is suspended, it completes the current partition, and stops sending science telemetry. Upon a Resume command, ensure that the resumed data starts at the beginning of a partition. For BINNED mode runnning a Burst Trigger variant, ensure that suspend and resume do not stop the burst trigger analysis.

g. Measure event processing performance.

h. Measure partition throughput performance, assuming an infinite telemetry resource.

i. Ensure that, when an observation is configured to govern its telemetry, that no more than one science data packet is sent per MajorTick.

Appendix A

A.1 Items Under Review

This section lists items and issues which need to be resolved:

Document Reference	Section	Resolution Expected by
NONE		

A.2 Patching Post-Launch Science Modes

The Flight Software is patchable after launch. This facility may be used to install new science modes. The procedure for developing a new mode is not formally defined at this stage, but may include the following steps:

a. Develop CPU object code and DSP executable for the new mode. This code should allocate a new mode id and object dump parameters, using values not already in use by other modes.

b. Incrementally link the new CPU object code with the Flight Software memory map, to form a patch image. The object code should be located somewhere in the PRAM memory space of the CPU.

c. Develop a patch for the Science Thread's dispatch table.

d. On ground hardware, store the CPU and DSP code and data into the proper locations in PRAM.

e. On ground hardware, install the patch into PRAM, reset and test.

f. Repeat steps (c) - (e) on the in-flight hardware.

For a mode that is used only once, the installation of the code, data, and patched dispatcher can be performed by first disabling science on one of the EA's, patching the EA's RAM directly, using Write EA Memory commands, and starting the mode by using the Call Subroutine command.