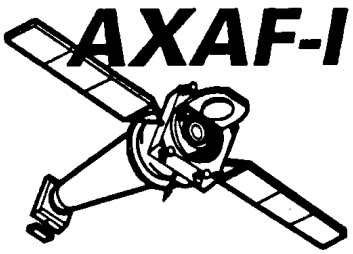
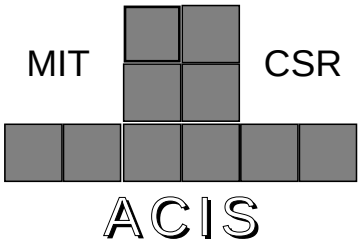
36-53200 Rev. 01++
February 3, 2000
NAS8-37716
DR SDM03

CSR





Advanced X-ray Astrophysics Facility

AXAF - I CCD Imaging Spectrometer

ACIS Science Instrument Software Detailed Design Specification (As-Built)

Submitted to:

George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

Submitted by:

Center for Space Research
Massachusetts Institute of Technology
Cambridge, MA 02139

AXAF-I CCD Imaging Spectrometer
(ACIS)

ACIS Science Instrument Software Detailed Design Specification (As-Built)

36-53200 Rev. 01++

DR SDM03

Contract # NAS8-37716

February 3, 2000

Submitted to:

George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

Submitted By:

Massachusetts Institute of Technology
Center for Space Research
77 Massachusetts Avenue
Cambridge, MA 02139

Approvals:

Dr. Peter Ford
Software Project Manager
Massachusetts Institute of Technology

Dr. William Mayer
Project Manager
Massachusetts Institute of Technology

CSR	**MASSACHUSETTS INSTITUTE OF TECHNOLOGY** **CENTER FOR SPACE RESEARCH** **CAMBRIDGE, MASSACHUSETTS 02139**	
REVISION LOG	**TITLE:** **ACIS Science Instrument Software Detailed Design Specification (As-Built)**	**DOC. NO.** **36-53200 Rev. 01++**

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
01	2/3/00	36-277	All	Initial Release for Software CDR	

Table of Contents

List of Figures

List of Tables

ACIS Science Instrument Software Detailed Design Specification (As-Built)

MIT Center for Space Research

36-53200 Rev. 01++

February 3, 2000

1.0 Introduction (36-53201 01)

The AXAF-I CCD Imaging Spectrometer (ACIS) Science Instrument Software (SIS) is being developed by the Massachusetts Institute of Technology, Center for Space Research (MIT-CSR) as part of the ACIS Digital Processor Assembly (DPA). The DPA resides on-board the Advanced X-ray Astrophysics Facility - Imaging (AXAF-I). The DPA Science Instrument Software is responsible for acquiring and processing image data from the ACIS CCD Imaging Spectrometer and transferring the processed data to the AXAF-I Command and Telemetry Unit (CTU), which is then responsible for sending the information to the ground.

1.1 Purpose

The ACIS Science Instrument Software Detailed Design Specification (Code-To) describes the design of the instrument software in sufficient detail to permit code development.

1.2 Scope

This document applies to the detailed design of the ACIS DPA Science Instrument Software. It does not provide information for the Ground Support Software (GSS), which is maintained separately as part of the Electronic Ground Support Equipment (EGSE).

This document supplies information applicable to SDM03 from the original contract, and to DM09 from MM8075.1.

By mutual agreement, MSFC Software Management and Development Requirements Manual MM8075.1, which supersedes MA-001-006-2H, forms the basis for this document.

1.3 References

This specification relies on a set of existing documentation. The following table lists these documents.

TABLE 1. Reference Documents

Part Number	Version	Title
MSFC MM 8075.1	January 22, 1991	MSFC Software Management and Development Requirements Manual
MIT-CSR 36-01103	B	ACIS Science Instrument Software Requirements Specification
MIT-CSR 36-01502	04	ACIS Technical Analyses and Models: ACIS Hardware Specification and System Description
NU910701	1991	Nucleus RTX Reference Manual from Accelerated Technology, Inc.
NU910702	1991	Nucleus RTX Internals Manual from Accelerated Technology, Inc.
ISBN 0-8053-5340-2	1994	Object-Oriented Analysis and Design with Applications, Second Edition by Grady Booch, Benjamin/Cummings
NASA Reference Publication, 1319	September, 1993	Mongoose ASIC Microcontroller Programming Guide, Brian S. Smith, GSFC
ISBN 1-55860-297-6	1994	MIPS Programmer’s Handbook by Erin Farquahar and Philip Bunce, Morgan Kaufman Publishers
ISBN 0-13-584749-4	1989	MIPS RISC Architecture, by Gerry Kane, Prentice Hall
MIT 36-10410	TBD	ACIS Instrument Protocol and Command List
MIT 36-02205	A	DPA/DEA Interface Control Document

2.0 Assumptions and Conventions

2.1 Audience

This document assumes that readers will be familiar with the ACIS Contract End Item Specification, and the ACIS Science Instrument Software Requirements Specification.

2.2 Portability

This document assumes that there are no hardware nor operating system portability requirements on the instrument software design or implementation.

2.3 Implementation Language

This document assumes that the Back End Processor software design is implemented in C++, and that the Front End Processor design is implemented in C. Unless otherwise specified, all data types and example code shown in this document use C++ notation.

2.4 Compiler Selection

This design assumes that the GNU C++ compiler, running on an DECstation 3000 or 5000, is used to compile the flight version of this software. Other compilers may be used for unit and integration testing portions of the software, but there may be parts of the software which are compiler specific.

2.5 Graphic Notation

Unless otherwise specified, diagrams and detailed class descriptions presented in this document use Booch Notation, as described in “Object-Oriented Analysis and Design with Applications,” by Grady Booch, 1994.

Figure 1 illustrates the icons and associations this document uses to illustrate the relationships between class and structure definitions.

FIGURE 1. Class Diagram Icons

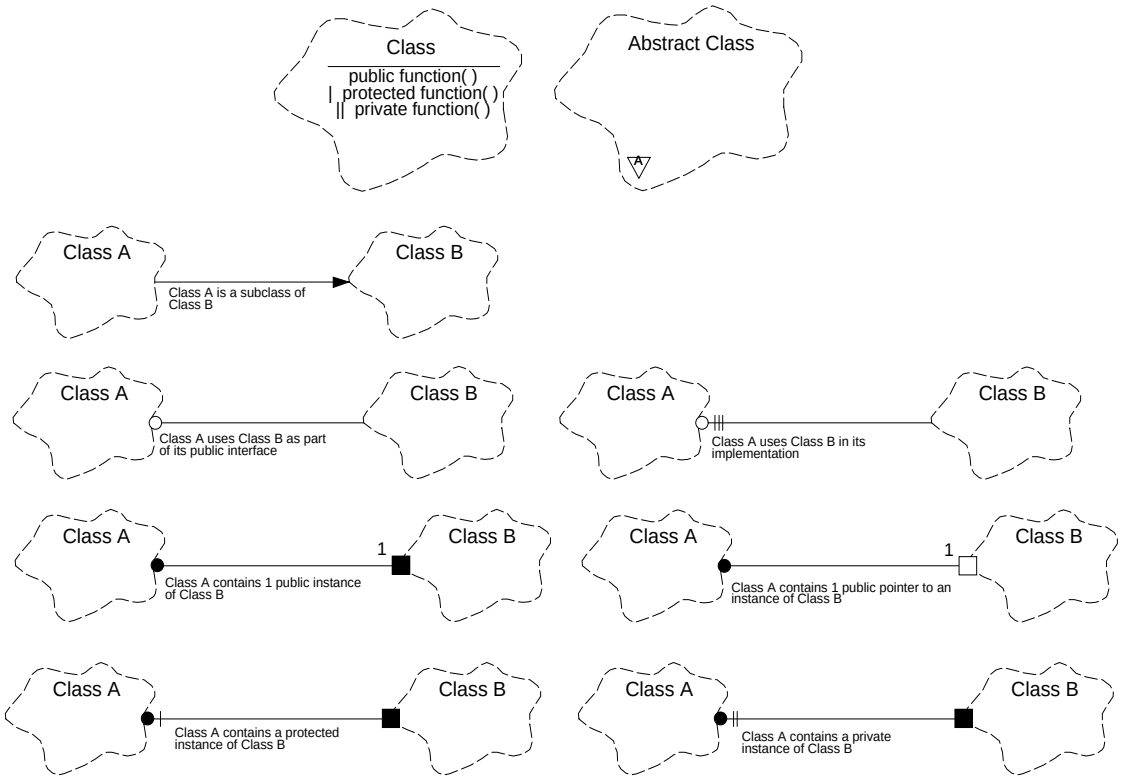
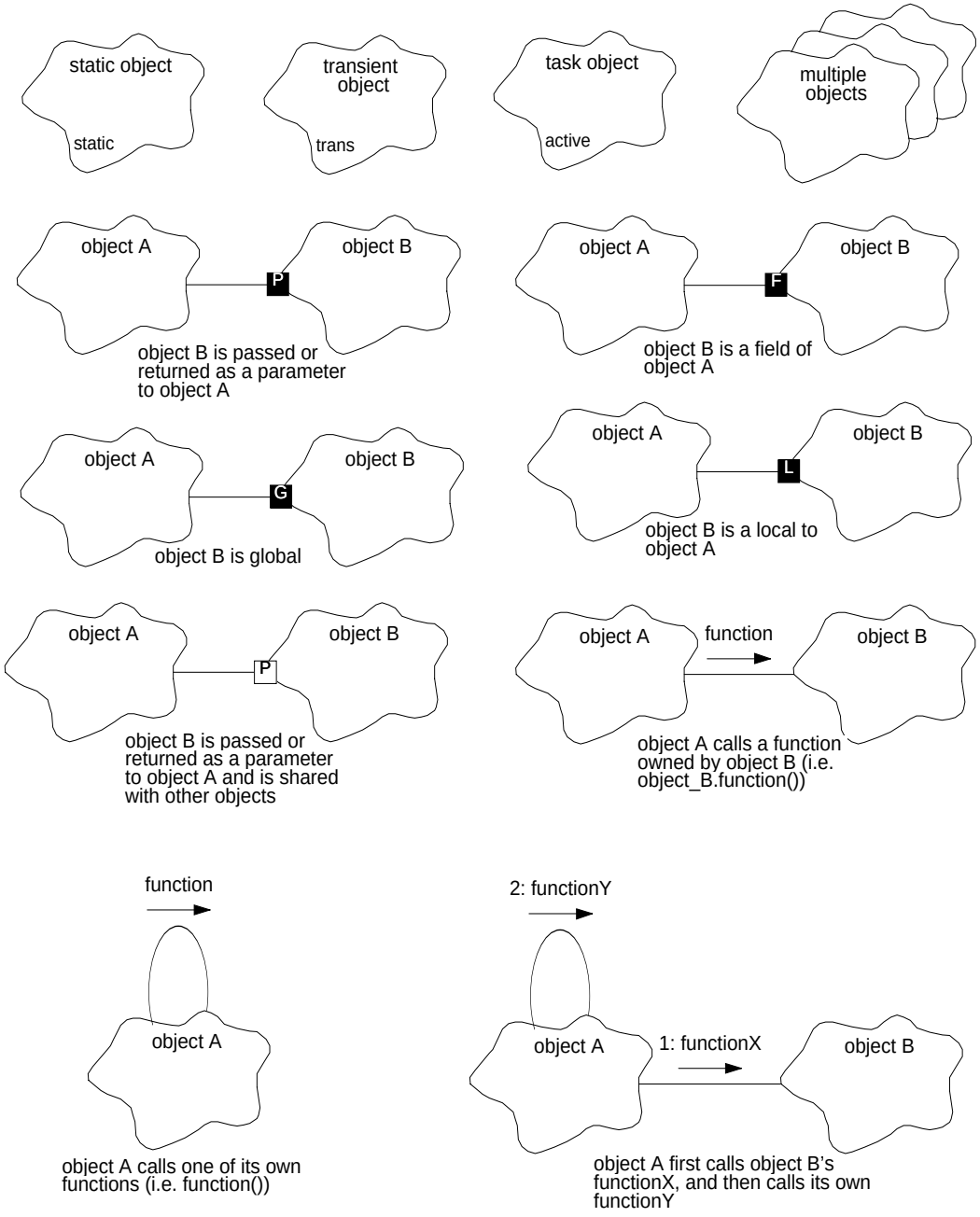


Figure 2 illustrates the icons and relationships this document uses to illustrate scenarios involving different objects (i.e. class instances).

FIGURE 2. Object Diagram Icons



2.6 Typographic and Naming Conventions

2.6.1 Class Category Names

All words and abbreviations of all class category names start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All names of class categories (i.e. subsystem) within this document are represented using bold, italicized Courier text:

`ClassCategoryName`

2.6.2 Class Names

All words and abbreviations in all class, structure, enumeration, and union names start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All class names within this document are represented using bold Courier text:

`ClassName`

2.6.3 Function Names

The first word or abbreviation in all function and member function names shall start with a lower-case letter. All remaining words, if any, within the name shall start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters. All function argument names shall conform to the convention described for Variable Names (see Section 2.6.4).

Functions which suspend execution of the currently running process until some condition is satisfied shall contain the word “wait” embedded in their name. Functions which require some condition, but do not suspend the process shall contain the word “request” embedded in their name. Functions which contain an infinite loop, such as the main function of a process, shall have the word “go” contained within their name.

All function names are represented using plain Courier text:

`functionName`

or

`functionName()`

2.6.4 Variable Names

The first word or abbreviation in all function and member function names Shall start with a lower-case letter. All remaining words, if any, within the name may or may not start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All variables, instance variables and structure or enumeration tags are represented using italicized Courier text:

variableName

2.6.5 Enumeration Tags

With the exception of the Boolean type, all letters in enumeration tag names are capitalized.

This document uses no special typographic convention for displaying enumeration tag values:

ENUMTAG (except for BoolTrue and BoolFalse, see Section 2.7)

2.6.6 Preprocessor Definitions

All preprocessor definitions are completely capitalized. Spaces may be indicated using underscores.

This document uses no special typographical convention for displaying preprocessor definitions:

PREPROCESSOR_DEFINITION

2.6.7 Directory and File Names

Except for filename extensions, all directories and filenames are in lower-case, with spaces represented as underscores. All C-language source and header files use the “.c” and “.h” extensions, respectively. All C++-language source and header files use the “.C” and “.H” extensions respectively. All assembler files have the “.s” extension.

This document uses no special typographical convention for displaying directory or filenames:

directory_name/filename.C

2.7 Global Data Types

2.7.1 Bit and Byte Ordering

Given that there are no portability requirements on the ACIS software, unless otherwise specified, it assumes that all data elements have little-endian byte ordering. For example, the 32-bit value 0x12345678 is stored as bytes in RAM as follows:

Virtual Address	Byte Value
0	0x78
1	0x56
2	0x34
3	0x12

By convention, bits within a word are numbered with the least-significant bit as bit number 0. This is consistent with the “MIPS Programmers Reference Guide.”

Unless otherwise specified, all signed values use two’s complement representation.

2.7.2 Integer, Pointer and Enumeration Data Types

Given that there are no portability requirements on the ACIS software, rather than provide renamed type definitions, the ACIS software assumes that the compiler defines its concrete data types as follows:

TABLE 2. Concrete Data Type Assumptions

C/C++ Type(s)	Sign and Size
char	signed 8-bit value
unsigned char	unsigned 8-bit value
short	signed 16-bit value
unsigned short	unsigned 16-bit value
int or long	signed 32-bit value
unsigned, unsigned int, or unsigned long	unsigned 32-bit value
all pointers	unsigned 32-bit values
enumerated types	unsigned 32-bit values

2.7.3 Boolean Data Type

In order to distinguish true and false arguments and return values from signed integers (int), the ACIS instrument software defines a **`Boolean`** data type using the following enumeration:

```
enum Boolean
{
   BoolFalse = 0,
   BoolTrue = 1
};
```

This type is used by the software whenever an argument or return type expresses a true or false expression. Since, in C and C++, enumerated types can be converted to an integer type by the compiler, these enumerated values are compatible with compiler generated relational expressions. NOTE: In C++, relational expressions, however, can not be converted to a **`Boolean`** type without an explicit cast or conversion.

For example:

```
Boolean result = BoolTrue;
if (result == (3<4))      /* ok, result is compared as an int */
{
}
result = (3<4);           /* wrong, int not converted to Boolean */
result = (Boolean) (3<4); /* ok, explicit type cast */
```

2.8 Global Units

2.8.1 ACIS Timestamp Units

The ACIS timestamp counter relies on the spacecraft-supplied 1.024 MHz clock. As such, all timestamp values are in units of 0.9765625 microseconds.

2.8.2 Timer Tick Units

All timer tick references made within this document are in units of 100 milliseconds (the Back End Processor timer-tick period).

2.9 Nomenclature

2.9.1 Classes and Objects

In object-oriented design, there are “objects” and “classes.” A class refers to the equivalent of a data structure definition and a collection of functions which operate on this data structure. An object refers to a physical declaration of a class. For example, in “C” one can have a structure such as:

```
struct foo
{
    int a;
};
```

`struct foo` is analogous to a class definition where as in declaration of one of these structures, such as in the case of:

```
struct foo bar
```

`bar` is analogous to an object declaration.

Within this document, the **Architecture** section describes the system primarily in terms of specific objects. Later portions of this document develop and describe the various classes within the Back End Processor software, and focus on the detailed behaviors of these classes.

2.9.2 Class Instance Lifetimes

The class descriptions in this document specify the lifetime of an instance of the class using two keywords:

- *Persistent* - Instances of this class exist for the lifetime of the program (i.e from startup to shutdown)
- *Transient* - Instances of this class may be created and destroyed as the program runs.

The use of the word *persistent* is a little misleading, in that it implies that the state of an object may be retained across instrument resets. Although there are a few such items, this document uses the keyword to indicate objects which are created at startup, and which last until the instrument is reset. All instances of a *persistent* class must last for the duration of the program. *Transient* objects are those which are created and initialized as needed by the running program, and destroyed when they are no longer in use. Some objects defined from *transient* classes may exist for the lifetime of the program, and others may come and go as the program runs. Unlike most C++ programs, the ACIS instrument software does not use a global run-time memory heap in order to avoid heap fragmentation and other issues. The memory for *transient* objects is obtained from collections of memory buffers, or on the stack. The lifetime of an object allocated from a memory pool ends when the object’s destructor is invoked, and its memory is released back to its pool. The lifetime of

an object declared on the stack ends when the scope of the code block which declared it ends.

2.9.3 Class and Function Concurrency

The class and function descriptions in this document specify the type of task and interrupt environment supported by the class or function, using three keywords:

- *Sequential* - Correct operation is guaranteed if only one task has access to the class or function
- *Guarded* - Correct operation is guaranteed if multiple tasks coordinate access to the class or function
- *Synchronous* - The class or operation performs all operations needed to coordinate access by multiple tasks. No special operations are required by the client.

In general, classes and functions used only during initialization, assume that only one thread of control is active, and specify *Sequential*. Classes and functions which expect to be used by only one task, or expect multiple tasks to arbitrate amongst themselves for access to the class or function, specify *Guarded*. Classes and functions which either do not require any inter-task arbitration (i.e. do not directly or indirectly modify any shared variable or hardware), or perform the arbitration internally, specify *Synchronous*.

3.0 Architecture (36-53203 A)

3.1 Purpose

This section describes the overall architecture of the ACIS Science Instrument Software, and identifies the key interfaces between the hardware and the software, and between the major software components.

3.2 Overall Approach

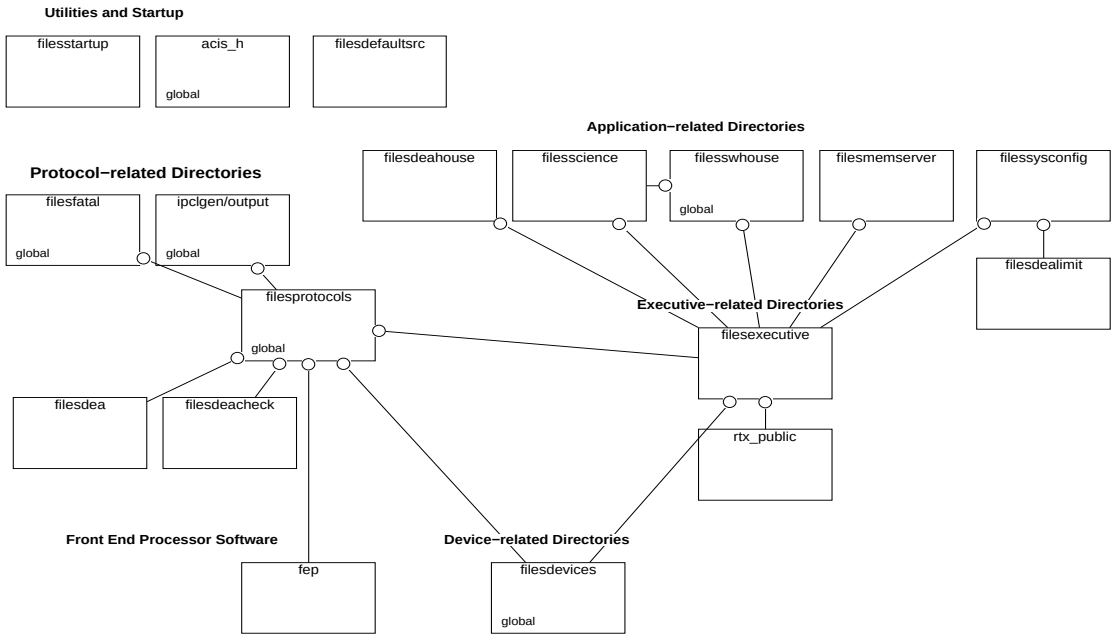
The bulk of the ACIS Science Instrument Software resides in the Back End Processor (BEP). This software is responsible for managing commands and telemetry, and for filtering and packing science data. The design of the BEP software employs object-oriented techniques, and uses a commercial real-time multi-tasking executive.

The Front End Processor (FEP) software is responsible for processing raw pixel data as quickly as possible. In general, this software has to deal with only one task at a time. As a result, its design employs a simple interrupt handler and single main thread of control, and takes a structured design approach.

3.3 BEP Class Categories and Source Code Directories

Class categories are a way of grouping related classes. Early in the design, categories were used to logically separate the layers of the design. As the design evolved, the source code directory structure superceded this layering to some extent, although some of the aspects of the layered architecture remain. The following illustrates the current BEP software class category directories (NOTE: ***ipclgen*** resides at the same level as the other class cateories):

FIGURE 3. Class Category Directories



Early in the ACIS design, there were five main categories. As the design evolved, it was discovered that more fine-grained categories were needed to group more tightly related classes. The original category set was as follows:

- ***Devices*** - These classes deal with the physical ACIS hardware. This class category remains intact within the current design. All source code within this category is stored in the ***filesdevices*** directory.
- ***Executive*** - These classes provide an interface to the underlying real-time executive. This class category remains intact within the design. All source code within this category is stored in the ***filesexecutive*** directory.
- ***Protocols*** - These classes originally dealt with externally imposed protocols, such as command packet formats, telemetry formats, etc. Since the original design, however, the classes within this category have focused more on buffer management, protocols, and high-level software interfaces to the key external components, such as the Front End Processors, the Detector Electronics Assembly, and the command and telemetry system. All source code for this category is stored in the ***filesprotocols*** directory.

  This class category contains within it, a ***Command Handler*** class category, used to group all of the command handler classes within ACIS. The source code for the command handlers is stored in the ***filesprotocols/filescmdhandlers*** directory (not shown).

All command and telemetry packet formatting, except for fatal error message formatting, is split off into a separated but related category which contains code generated from the ACIS Software Instrument Program and Command List (IP&CL) Structures (MIT 36-53204.0204) definitions. The scripts used to translate the IP&CL into source code are in the ***ipclgen*** directory. The generated code is in the ***ipclgen/output*** directory.

The responsibility for the fatal error telemetry packet formatting class remains in the ***Protocols*** category. However, the class responsible for issuing the fatal message and resetting the Back End Processor is in its own category, whose source is in the ***filesfatal*** directory.

During the design of the software, the Detector Electronics Assembly (DEA) command protocols evolved to the point that a complete set of classes were developed to implement the protocol. These classes are contained in the ***filesdea*** directory. It was discovered that the DEA sequencers also require a checker to ensure that the execution of the DEA memory loads does not overheat parts of the circuit. This responsibility is provided by the classes in the ***filesdeacheck*** directory.

- ***Applications*** - Originally, these classes dealt with the implementation of the core system requirements, such as performing a science mode, or dumping the contents of memory. Each application, however, became sufficiently detailed, and isolated from the other applications that each major feature is contained in its own category and source code directory.

  The responsibility for acquiring and telemetering housekeeping information from the Detector Electronics Assembly is provided by the classes in the ***filesdeahouse*** directory.

  The ***filesswhouse*** directory provides the source code for accumulating and telemetering software housekeeping statistics information.

  The ***filesmemserver*** directory provides services to read and write RAM within the instrument, and to execute subroutines in contained in the Back End Processor or the Front End Processor's RAM.

  The system's configuration table is maintained by the classes in the ***filessysconfig*** directory. Limit checking of certain DEA settings stored in the configuration table is provided by software in the ***filesdealimit*** directory.

  Finally, all of the many modes of science configuration software, and data processing software is contained in the ***filesscience*** directory.

- ***Utilities*** - These are classes which do not easily fit into one of the above categories. There are three class categories which fit this definition.

  The ***acis_h*** directory contains the main global constant definitions for the system, and the interface constants needed by the users of the instrument for commanding and telemetry.

  The ***filesboot*** directory contains boot-strap loader code, used by the Back End Processor to load code into its RAM, either from the BEP's Read-Only Memory, or from the command channel.

  The ***filesstartup*** directory contains the software used to initialize, patch and start the main Back End Processor software.

The ***filesdefaultsrc*** directory contains the data files, scripts and generated default tables used by the Back End Processor software.

Finally, all of the Front End Processor software is contained in the ***fep*** directory.

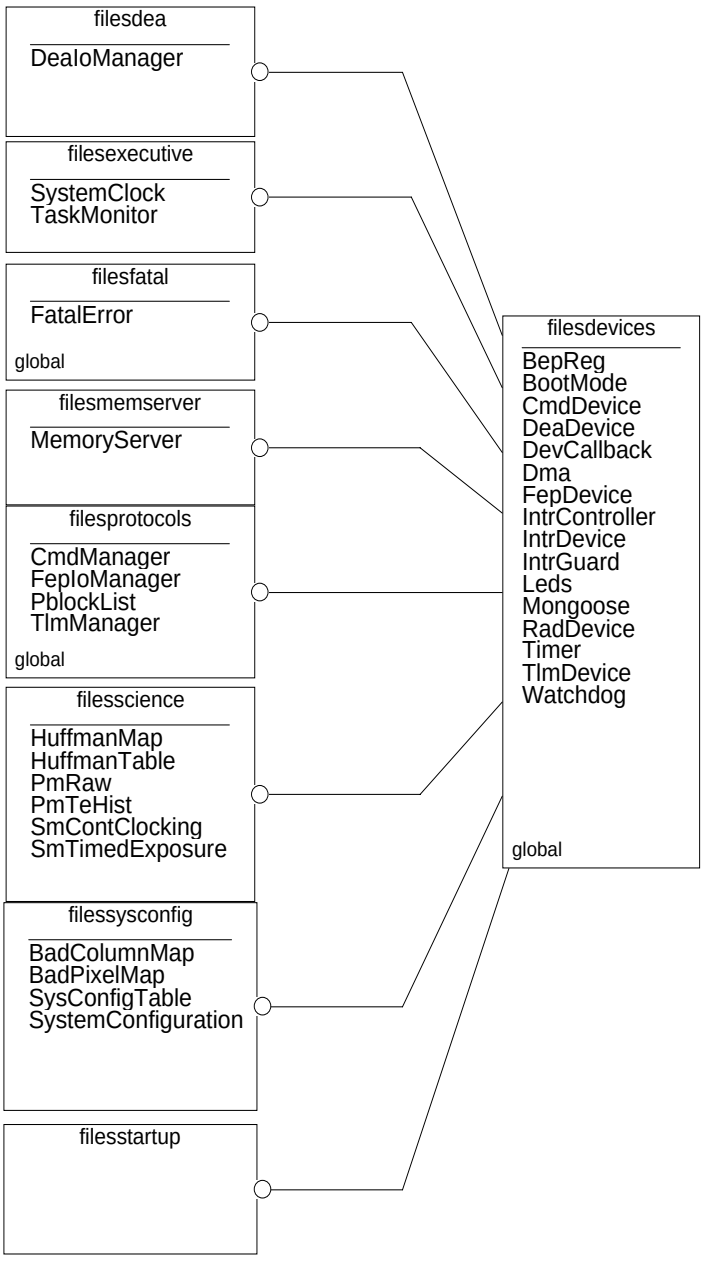
3.4 BEP Class Category Contents and Relationships

This section illustrates classes contained in each of the main category directories, and the clients and servers of those classes.

3.4.1 Device Classes

The device classes are responsible for directly interacting with the BEP hardware. The detailed design of the BEP device classes is contained in Section 5.0 through Section 12.0 .

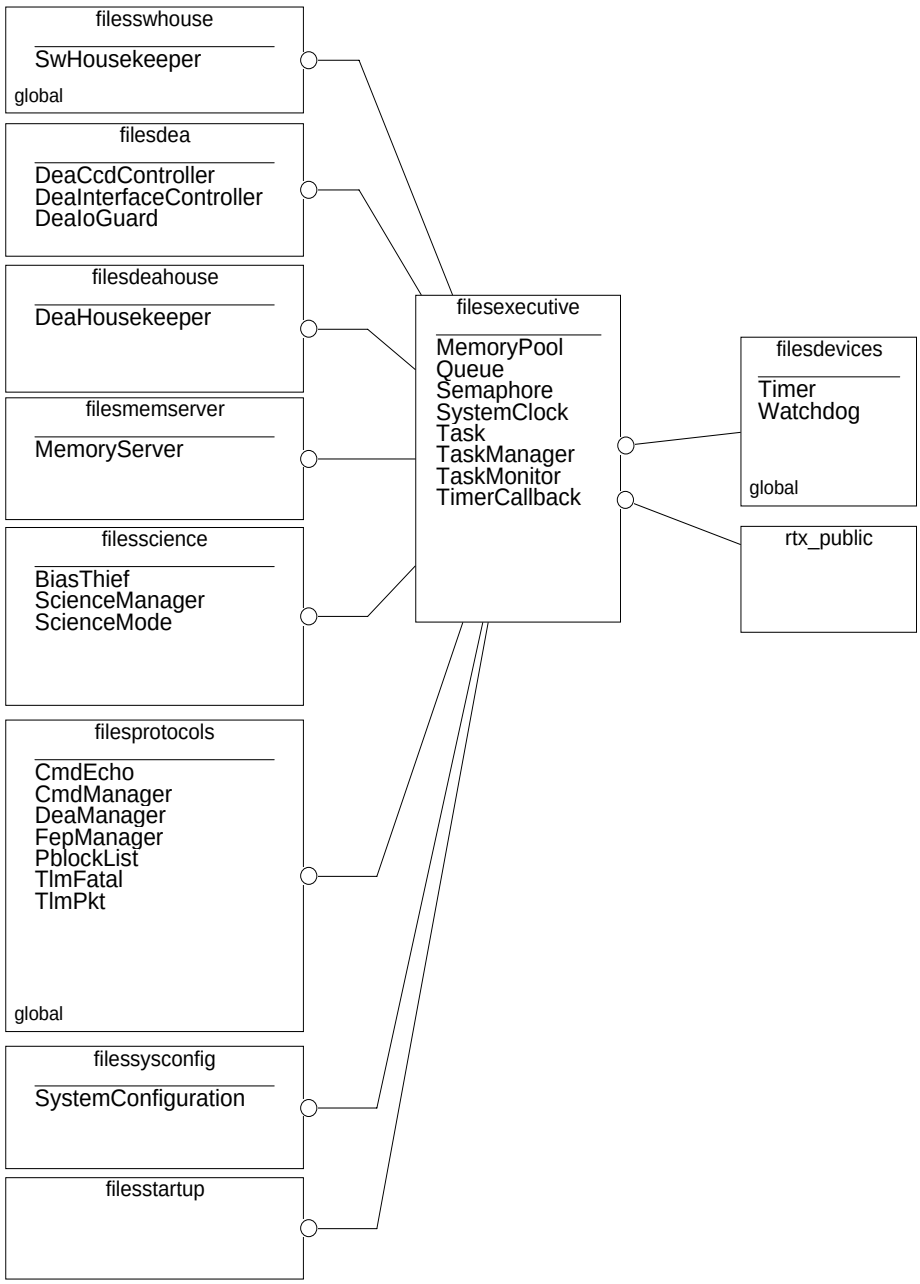
FIGURE 4. Device Class List



3.4.2 Executive Classes

The Executive classes are responsible for providing an interface layer between the main BEP software and the Nucleus RTX executive. The detailed design of the executive classes is described in Section 15.0

FIGURE 5. Executive Class List



3.4.3 Protocol Classes

The Protocol classes are responsible for managing a variety of interface protocols.

FIGURE 6. Protocol Class List

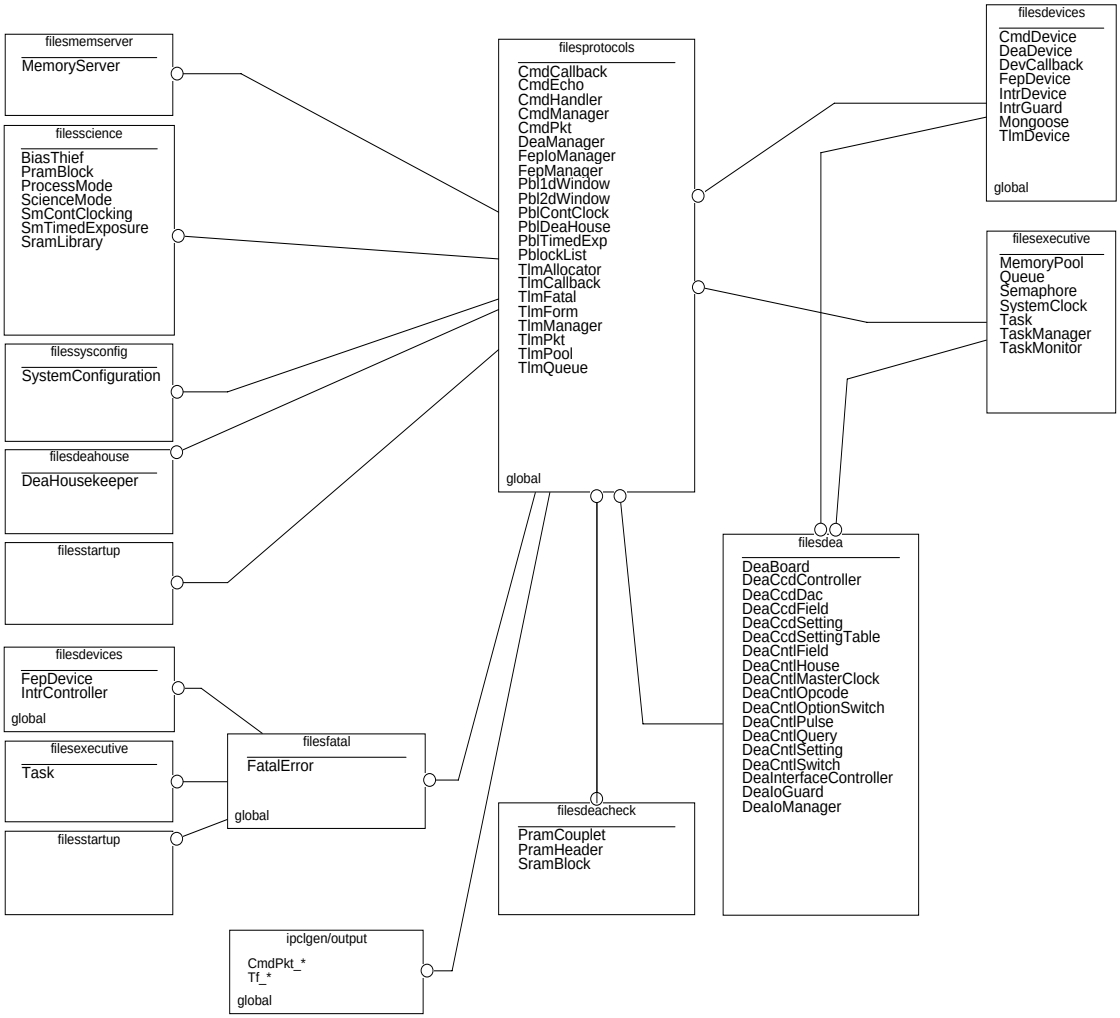
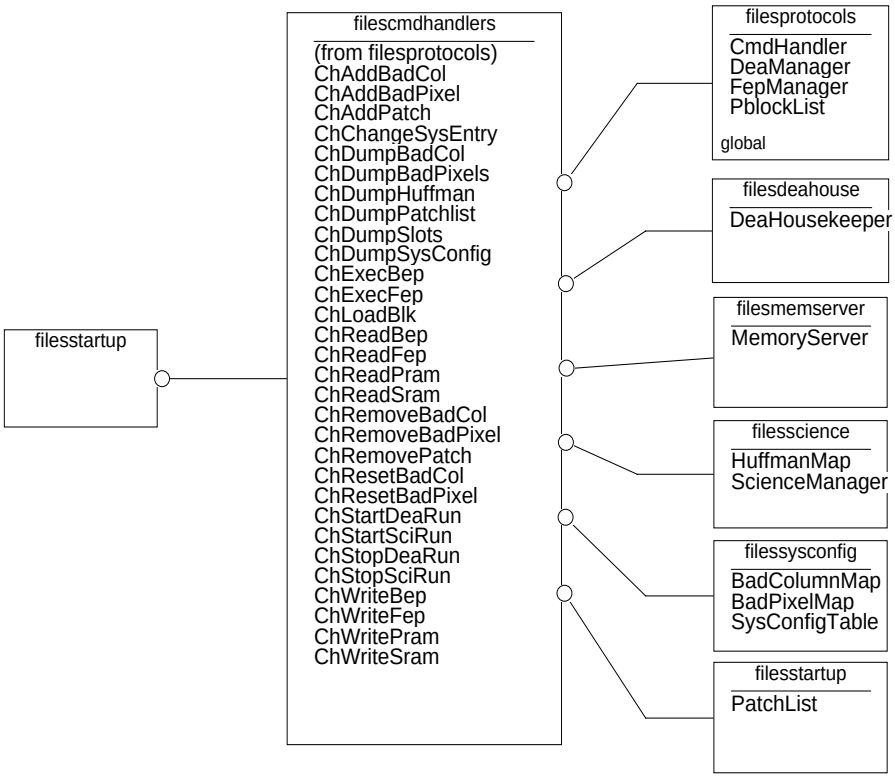


FIGURE 7. Protocols Command Handler Class List



The detailed design is segmented into sections as follows:

Command Management - Section 16.0 covers

CmdCallback
CmdEcho
CmdHandler
CmdManager
CmdPkt

Command Handlers - Section 17.0 covers all of the command handler classes (see Figure 7).

Telemetry Management - Section 18.0 covers

TlmAllocator
TlmCallback
TlmFatal
TlmForm
TlmManager
TlmPkt
TlmPool
TlmQueue

FEP Management - Section 25.0 covers

`FepManager`
`FepIoManager`

DEA Management - Section 26.0 covers

`DeaManager`

NOTE: Classes provided in ***filesdea*** and ***filesdeacheck*** have yet to be described. Section is TBD.

Parameter Block Management - Section 20.0 covers

`PblockList`

NOTE: **`PblTimedExp`**, **`PblContClock`**, **`PblDeaHouse`**, **`Pbl2dWindow`** and **`Pbl1dWindow`** have yet to be described. Section is TBD.

Fatal Error Reporting - Section 29.0 covers

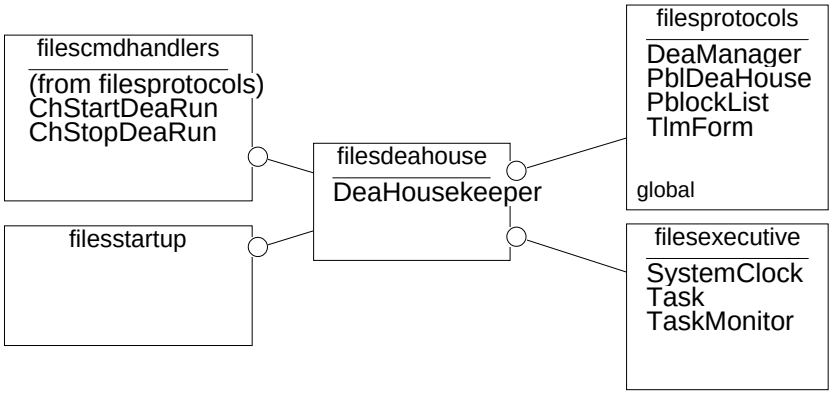
`FatalError`

IP&CL Code-Generation - Section 21.0 , Section 22.0 , and Section 23.0 cover
Code-generation scripts
Standard for Command Format Classes (**`CmdPkt_*`**)
Standard for Telemetry Format Classes (**`Tf_*`**)

3.4.4 DEA Housekeeping Classes

The DEA Housekeeper class is responsible for periodically acquiring and telemetering information from the Detector Electronics Assembly. Its detailed design is provided in Section 31.0

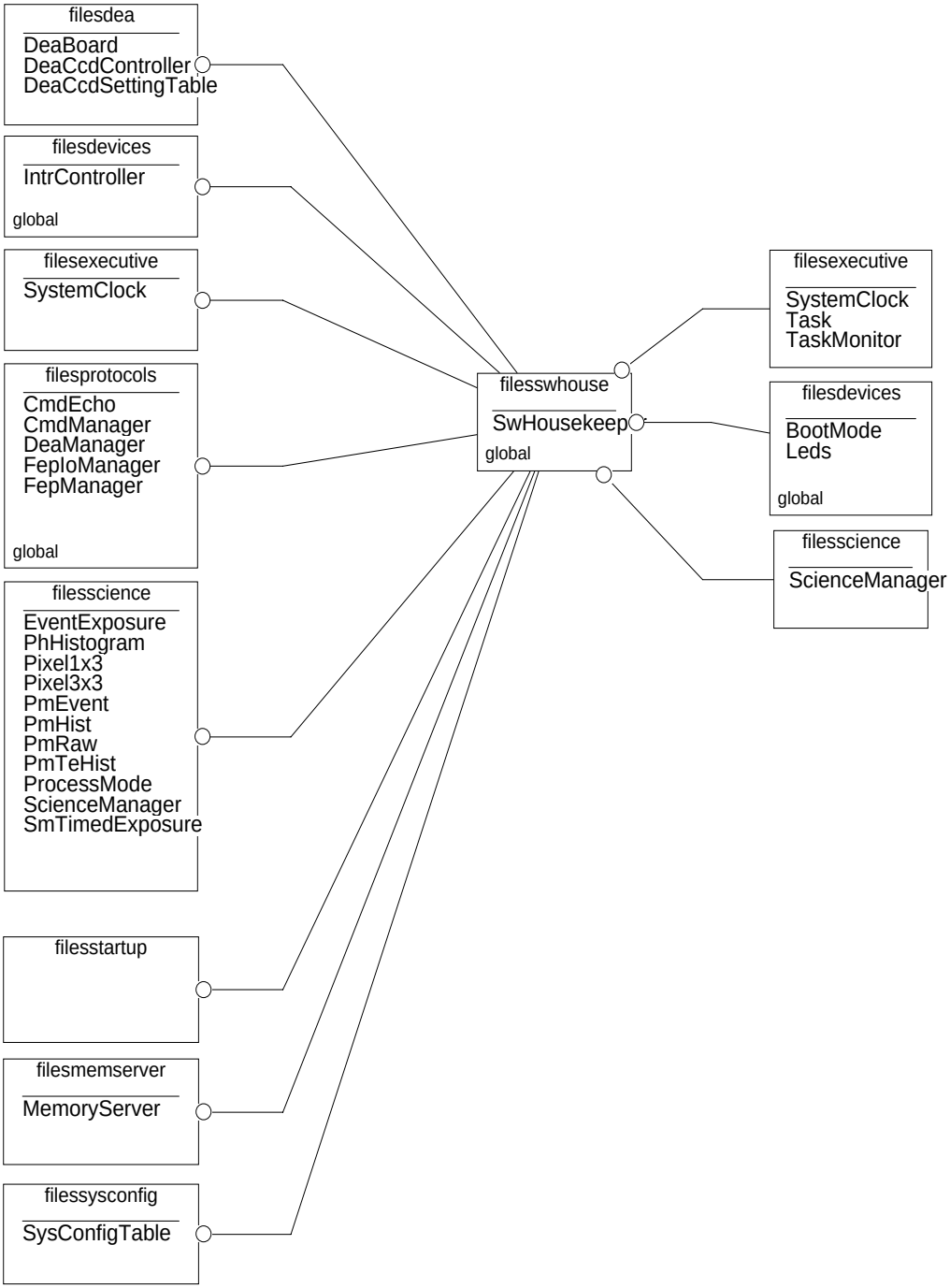
FIGURE 8. DEA Housekeeping Class List



3.4.5 Software Housekeeping Classes

The Software Housekeeper class is responsible for accumulating software housekeeping statistics reported by other software units within the BEP, and periodically telemetering the accumulated data. Its detailed design is described in Section 28.0

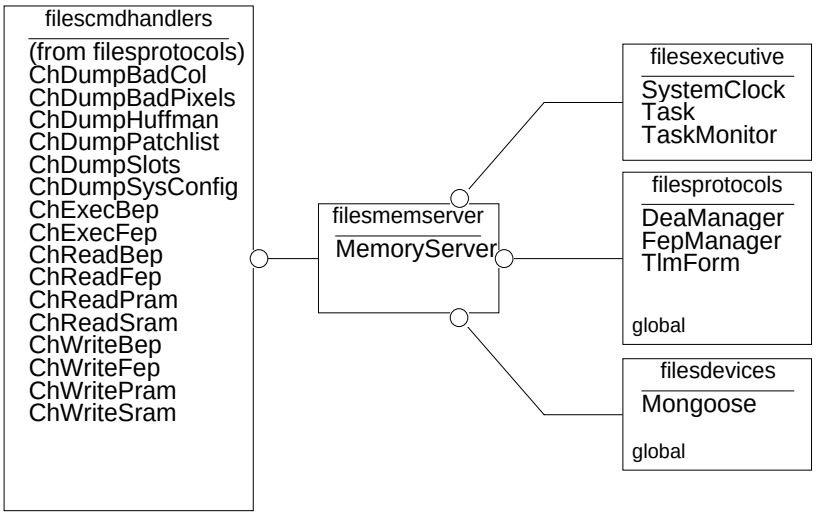
FIGURE 9. Software Housekeeping Class List



3.4.6 Memory Server Classes

The Memory Server class is responsible for servicing command requests to read (dump) or write portions of the BEP's, FEP's or DEA's memory, and to service command requests to execute code on the BEP or FEP. It uses the FEP Manager and DEA Manager classes to respectively forward requests to the FEP and DEA, when needed. Its detailed design is described in Section 27.0

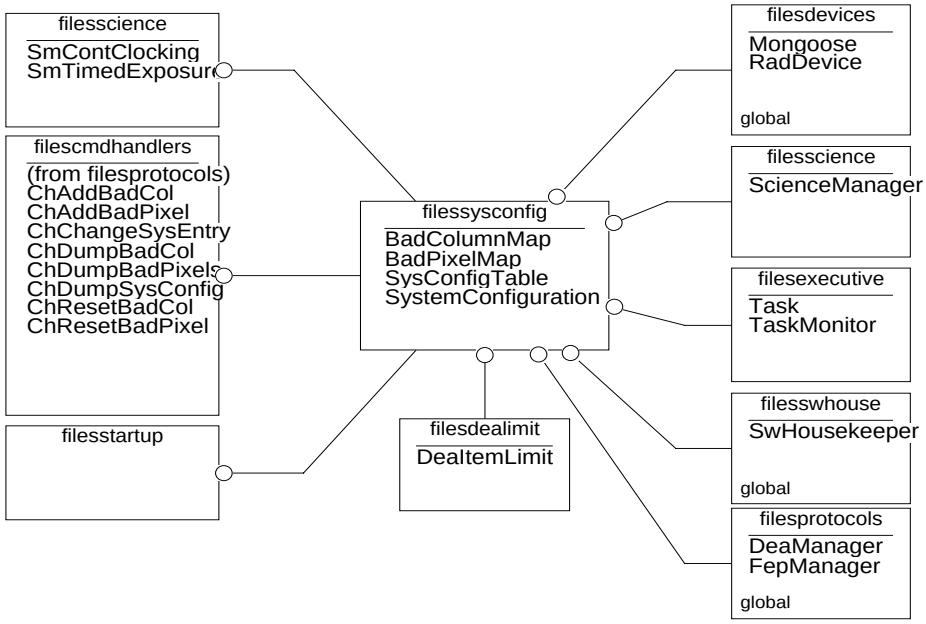
FIGURE 10. Memory Server Class List



3.4.7 System Configuration Classes

The System Configuration classes are responsible for maintaining the system configuration table, and the Bad Pixel and Column maps.

FIGURE 11. System Configuration Class List



The detailed design of the classes are located as follows:

System Configuration Table Management - Section 30.0 covers

SysConfigTable
SystemConfiguration

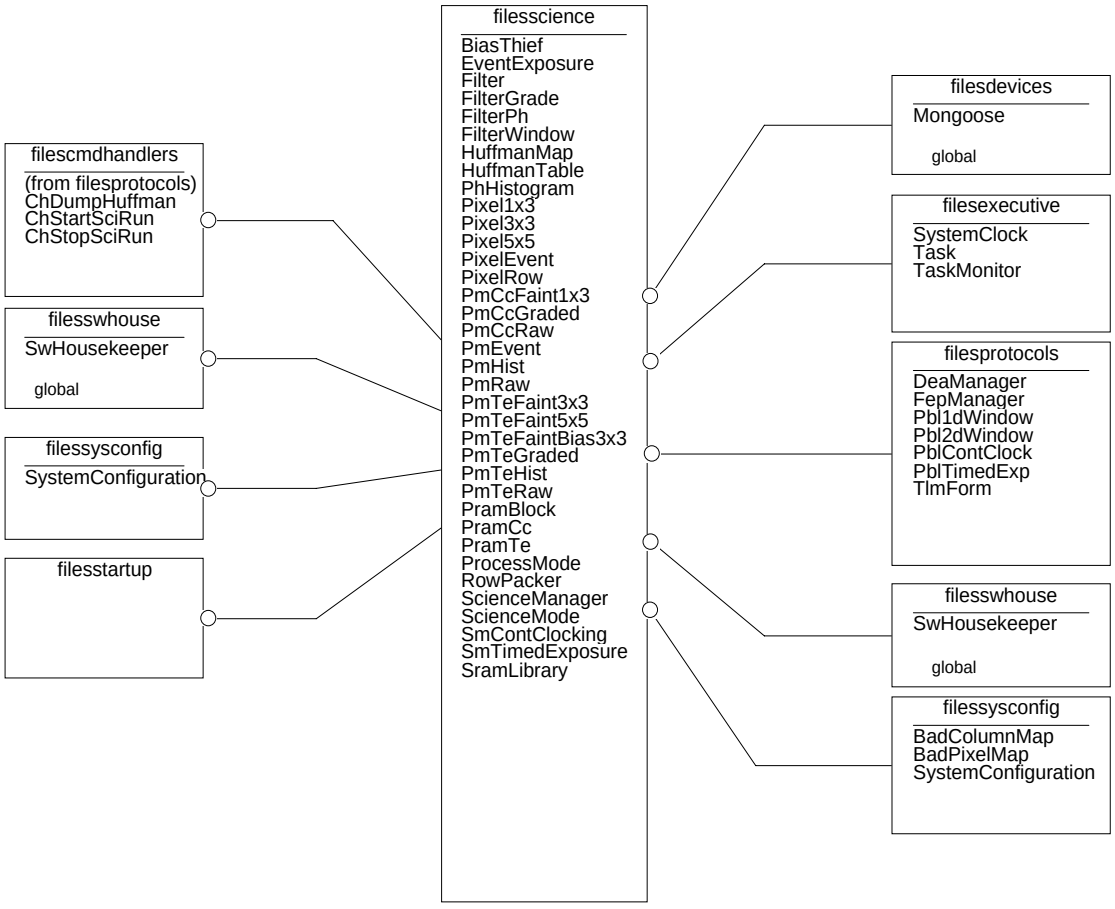
Bad Pixel and Column Map Management - Section 32.0 covers

BadColumnMap
BadPixelMap

3.4.8 Science Classes

The Science Management classes are responsible for a variety of activities involved in performing a science run, including run setup and execution and bias-map transmission.

FIGURE 12. Science Class List



The detailed design for the science classes is broken into several sections, as described below:

Science Management - Section 33.0 covers

ScienceManager
ScienceMode
ProcessMode

Science Data Processing - Section 37.0 covers

EventExposure
Filter
FilterGrade
FilterPh
FilterWindow
PhHistogram
Pixel1x3
Pixel3x3
PixelEvent
PixelRow
PmCcFaint1x3
PmCcGraded
PmCcRaw
PmEvent
PmHist
PmRaw
PmTeFaint3x3
PmTeFaintBias3x3
PmTeGraded
PmTeHist
PmTeRaw
SmContClocking
SmTimedExposure

NOTE: **Pixel5x5** and **PmTeFaint5x5** have yet to be described. Section is TBD.

Bias Map Telemetry Management - Section 38.0 covers

BiasThief

Huffman Data Compression - Section 24.0 covers

HuffmanTable

NOTE: **HuffmanMap** is not yet described. Section is TBD

SRAM/PRAM Setup - Section 36.0 , Section 34.0 and Section 35.0 cover

SramLibrary
PramBlock
PramTe
PramCc

3.5 BEP Tasks

The Back End Processor runs a preemptive, multi-tasking executive. During BEP start-up (see Section 14.0), all of the system's tasks are started. Once started, these tasks never exit. Tasks of a given priority are allowed to preempt tasks of a lower priority, and tasks of the same priority run until they are either preempted, or until they sleep for some period of time or relinquish control, at which point another task of the same priority is permitted to run. Once all tasks of a given priority are blocked, waiting for an event to occur, tasks of a lower priority are allowed to run.

The architecture of the BEP relies on a set of concurrently running tasks. Each task is represented by an object of a specific sub-class of the Task class. The following lists the BEP's tasks, listed and grouped according to their priority (highest priority, 51, is first, and lowest priority listed, 55, is last. Tasks with the lowest priority number have the highest run-time priority):

TABLE 3. BEP Tasks

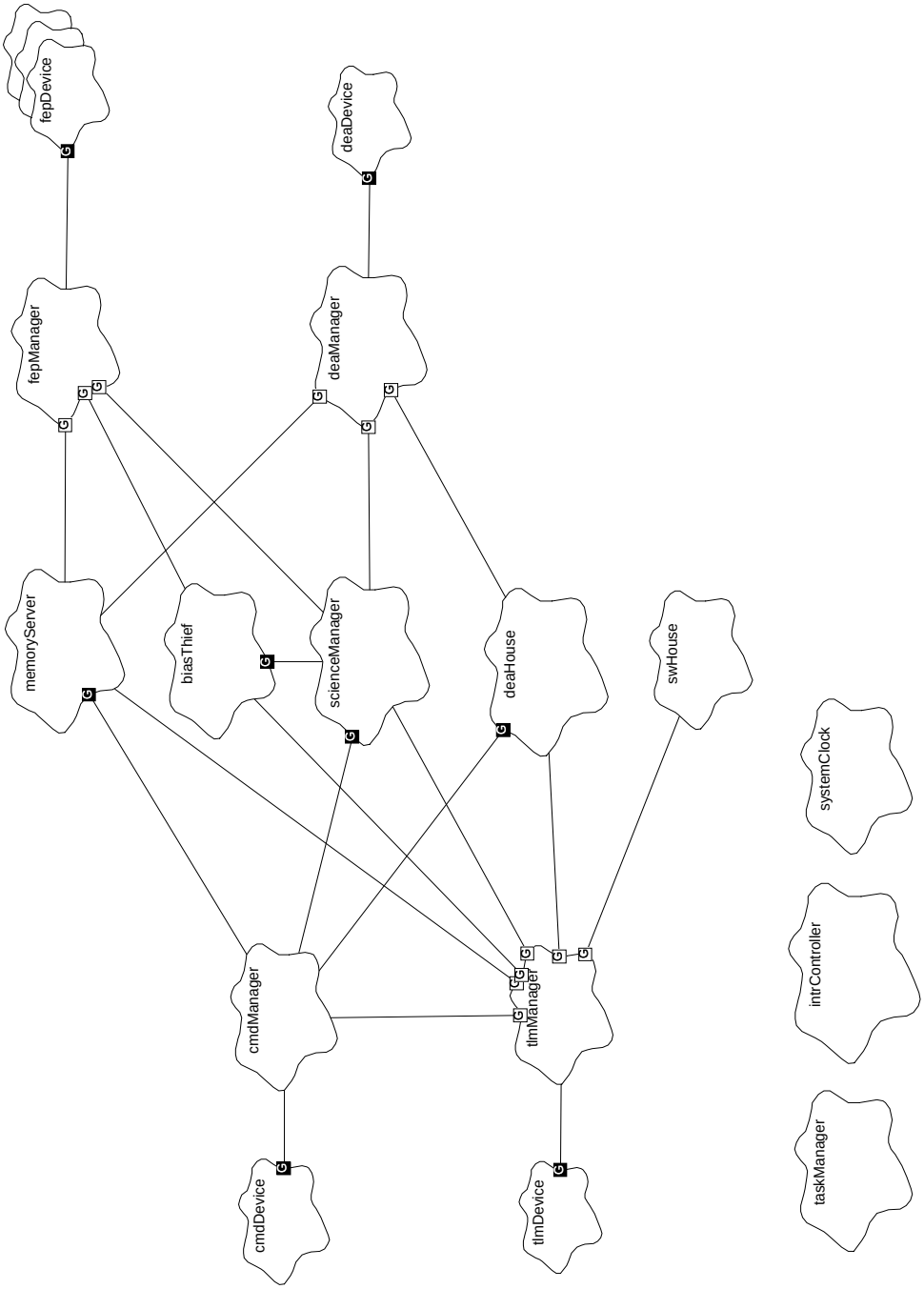
Class Name	Pri.	Object Name	Role
TaskMonitor	51	*taskMonitor*	Perform aliveness tests of the other tasks. Allows the watch-dog timer to reset the BEP if a task fails to respond to a query within 8 minutes.
CmdManager	52	*cmdManager*	Execute uplinked commands
SystemConfiguration	53	*systemConfiguration*	Respond to changes in configuration table and monitor the radiation flag
SwHousekeeper	53	*swHousekeeper*	Collect and periodically send software statistics. Update LED bi-levels to reflect instrument's operating state.
DeaHousekeeper	53	*deaHousekeeper*	Periodically collect and send DEA housekeeping values.
MemoryServer	54	*memoryServer*	Handle read (including dump), write, and execute memory commands
BiasThief	55	*biasThief*	Trickle the contents of the computed CCD bias maps to telemetry.
ScienceManager	55	*scienceManager*	Perform science run, including hardware setup, parameter dumps, bias computation and data processing

Each BEP task class provides two sets of member functions. One set is visible to clients of the class and may be called directly by any thread of control. In this document, these are known as "binding" functions. The second set of functions are internal to the task class, and must be called only by the task object's thread of control.

3.6 BEP Global System Objects

This section identifies the some of the objects within ACIS which are globally visible to the rest of the system. Figure 13, “BEP Global System Objects,” on page 70 illustrates the key higher level global objects within the ACIS software. In that figure, the solid-lined “clouds” represent objects, and the connecting lines show who is talking to whom. The filled boxes indicate that the object is exclusively used by the party at the other end of the line. The empty boxes indicate that the adjacent object is shared by several other objects.

FIGURE 13. BEP Global System Objects



Devices

cmdDevice - This object is responsible for physically reading command packets from the BEP's command hardware.

tlmDevice - This object is responsible for setting up the telemetry hardware to transfer the contents of a region of memory to the RCTU serial telemetry port.

deaDevice - This object is responsible for writing commands to the DEA command port, and for reading status words from its reply port.

fepDevice[6] - Each object corresponds to a single FEP. These objects are responsible for accessing the FEP control hardware and for accessing memory-mapped hardware and software mailbox locations within the corresponding FEP.

Protocols

cmdManager - This object is responsible for acquiring commands from the *cmdDevice* and executing the commands. It uses the *tlmManager*, via a *cmdLog* object (not shown), to acknowledge the reception of commands and indicate their disposition.

tlmManager - This object is responsible for queueing telemetry transfer requests from the many telemetry sources within the system. The *tlmManager* uses the *tlmDevice* to instruct the hardware to physically transfer the telemetry items.

deaManager - This object is responsible for formatting and sending commands to the Detector Electronics Assembly, and for processing any acquired status information and data. This object uses the *deaDevice* to issue command and retrieve responses from the physical DEA hardware.

fepManager - This object is responsible for commanding all of the FEPs and for managing data being produced by the FEPs. This object uses all of the *fepDevice* and *fepIoManager* objects to send and receive information to and from the individual FEPs.

fepIoManager[6] (not shown) - These 6 objects are responsible for managing the I/O protocol to and from each of the Front End Processors. Each manager corresponds to a single FEP.

Applications

memoryServer - This object is responsible for performing memory dumps, run-time memory loads, and commanded function calls. It uses the *fepManager* to forward such requests to the FEPs and the *deaManager* to perform DEA memory loads and dumps. The

memoryServer uses the *tlmManager* to transfer memory dumps and send return values from function calls into telemetry.

scienceManager - This object is responsible for managing science data production, acquisition, and processing. It uses the *deaManager* to load and command the DEAs to clock the CCDs. It uses the *fepManager* to acquire the resulting science data. This object uses the *tlmManager* to place the produced science data into the telemetry stream.

deaHousekeeper - This object is responsible for acquiring and sending DEA engineering data to telemetry. It uses the *deaManager* to request and acquire specific housekeeping values from the DEA, and it uses the *tlmManager* to place the acquired housekeeping data into the telemetry stream.

swHousekeeper - This object is responsible for accumulating and reporting various software housekeeping statistics. It uses the *tlmManager* to place the acquired data into the telemetry stream. Most objects in the system will occasionally report information to this object.

biasThief - This object acts under the direction of the *scienceManager*, and is responsible for acquiring and sending bias map data from the Front End Processors as telemetry and processing resources permit.

General Purpose

taskManager - This object is responsible for coordinating the activities of all of the tasks within the BEP. This object has indirect access to every task within the BEP (not shown).

intrController - This object is responsible for managing interrupts within the BEP. It has access to every interruptible device within the BEP, and provides interrupt enable/disable services to the rest of the BEP software.

systemClock - This object is responsible for providing the current time, in units of BEP timer-ticks, to the other objects within the BEP.

In addition to the objects described above, the Back End Processor also uses a variety of global low-level hardware interface objects to manage access to the Back End's CPU and the attached hardware. These include the following:

mongoose - This object is responsible for coordinating access to the R3000 System Coprocessor register and to the Mongoose Command/Status Interface (CSI) registers.

bepReg - This object is responsible for coordinating access to the Back End Processor's Control, Status and Pulse hardware registers, and providing access to the Command FIFO, Downlink Transfer Control, and the Detector Electronics Assembly Command, Status and Microsecond Timestamp registers.

dmaDevice - This object is responsible for managing transfers using the Mongoose's Direct Memory Access (DMA) device.

timerDevice - This object is responsible for managing the Mongoose's General-Purpose Timer device.

watchdogDevice - This object is responsible for managing the Mongoose's Watchdog Timer device.

3.7 FEP Software Architecture

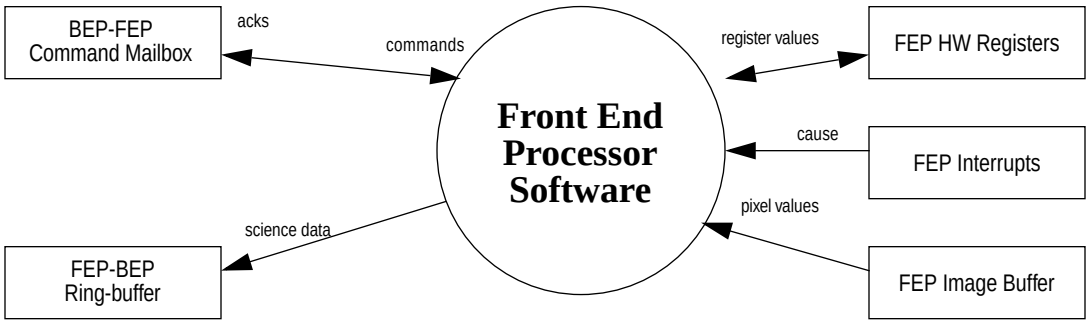
Within ACIS, there are six Front End Processors (FEP), each acting under the direction of the Back End Processor (BEP). During a given run, each FEP is responsible for processing images from one CCD.

This section summarizes the architecture of the software running on each of the Front End Processors. This software consists of two main types:

- Science Processing Functions
- Hardware and I/O Library

Figure 14 illustrates a simplified context diagram of the Front End Processor software.

FIGURE 14. Front End Processor Context Diagram



The interface between the Back End Processor and each of the Front End Processors is managed using shared-memory region residing on each FEP. The I/O library software running on a given FEP establishes and manages three interfaces with the Back End:

BEP-FEP Command Mailbox - This mailbox is used by the BEP to issue commands to the software running on a FEP, and to obtain the FEP's response to the command. This mailbox is primarily used by the BEP's **FepIoManager** class to load parameters, and start and stop bias and science activities on a FEP and to query FEP status.

FEP-BEP Ring-Buffer - This ring-buffer is used by a FEP to send large amounts of science data to the BEP. The data is organized into tagged data records, which are read by the BEP's **FepIoManager** class, subsequently parsed and processed by the BEP's Science Processing classes. A FEP primarily uses its ring-buffer to send exposure information records, and science data records, such as event records or histogram data records.

The I/O library software also provides low-level access functions to the Front End Processor hardware:

FEP Hardware Registers - Each FEP contains a set of hardware registers. These registers control the behavior of FEP's image acquisition and threshold hardware. The I/O library provides functions to read and write these registers. The BEP's **FepDevice** class access some of these registers across the shared-memory interface to reset the FEPs, and to determine the current reset state of the FEPs.

FEP Interrupts - Each FEP can be interrupted from a few sources. Its I/O library provides a common interrupt handler, which deals with all interrupt causes.

FEP Image Buffer - Each FEP contains a hardware-maintained image buffer, which is used to acquire CCD images for processing by the FEP software.

FEP Bias Map and Parity Plane (not shown) - Each FEP contains a hardware-maintained bias map buffer and parity plane, which is used by the FEP software to store CCD pixel bias values and verify the integrity of the bias map values. The BEP's **FepIoManager** class writes into this memory, via the shared memory interface, to mark bad pixels and columns. The BEP's **BiasThief** class reads from this memory when packing and telemetering the FEP's bias maps.

Figure 15 illustrates the overall data flow within the Front End Processors software. Shaded circles illustrate some of the services provided by the FEP Hardware and I/O library, and the unshaded circles illustrate the functions handled by the science processing functions.

The Science Processing functions perform three types of actions:

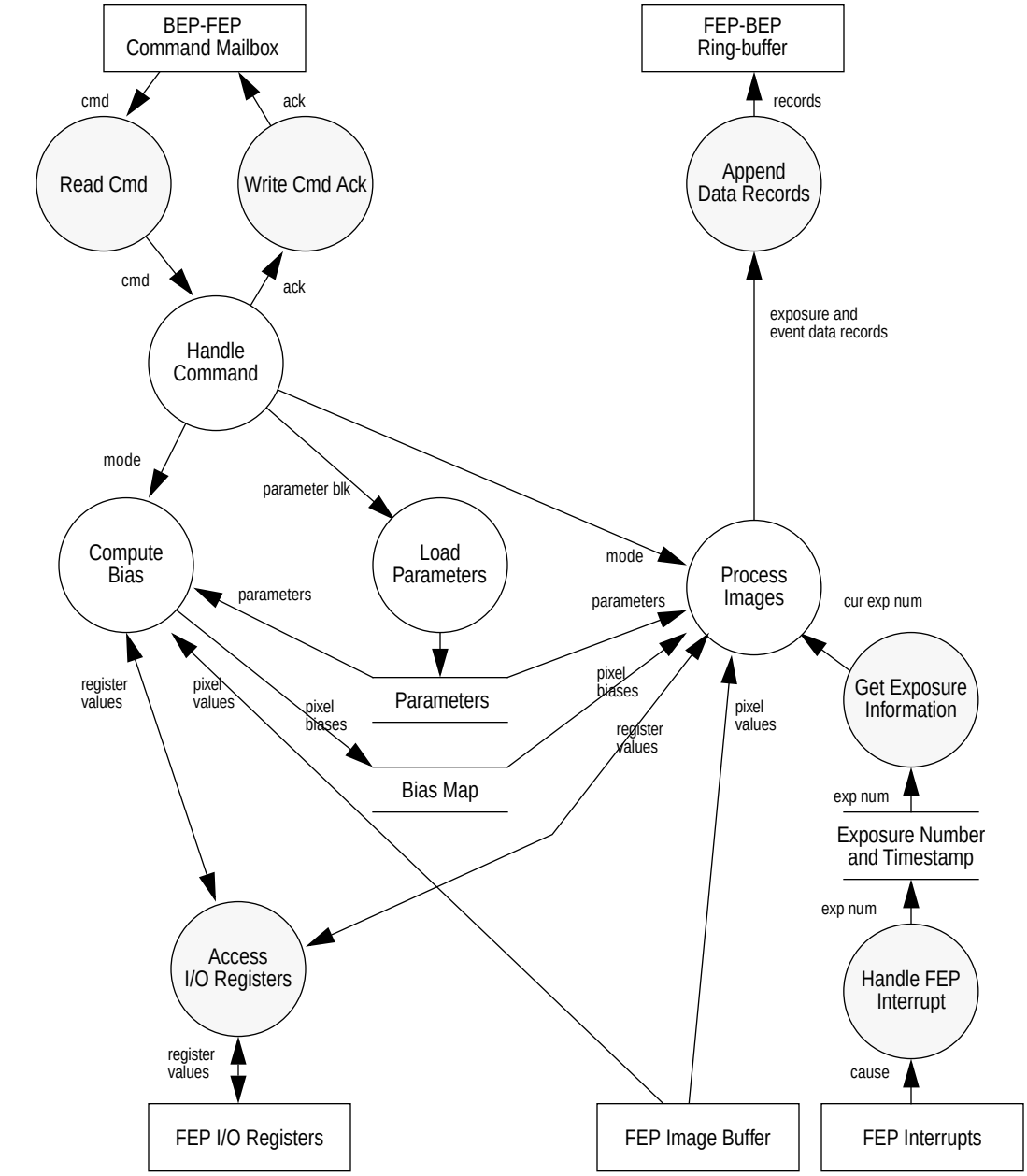
Load Parameters - This action is initiated by the Back End Processor, which passes parameters to each Front End using its Command Mailbox. The science software handles these commands between runs, storing the loaded parameters for use for the subsequent bias computations and data processing.

Compute Bias - The science software on a given Front End Processor is capable of computing the bias level for each CCD pixel represented in an image. This action is initi-

ated via a command from the BEP. The type of bias to perform, and the parameters to use for the bias computation, are provided by a previous `Load Parameters` action. The resulting map of pixel-by-pixel bias values is retained for use by subsequent data processing. NOTE: Although it is not shown in the diagrams, the bias maps are located in shared memory, and are visible to the Back End Processor. This enables the BEP to telemeter the contents of the maps. Unfortunately, due to unforseen timing issues in the hardware, access to this area during data processing interferes with the hardware event processing. As a result, the BEP software only accesses this memory prior to starting event processing on the FEPs.

`Process Images` - The science software provides an action which processes incoming images from a CCD. The BEP initiates and terminates this action via the Command Mailboxes, specifying which mode to use when processing the images. The parameters to use for data processing are provided by a previous `Load Parameters` action, and the pixel bias values used are those computed by the most recent `Compute Bias` action.

FIGURE 15. Front End Processor Data Flow Diagram



3.8 Functional Overview

This section provides an overview of the behavior of key features of the system.

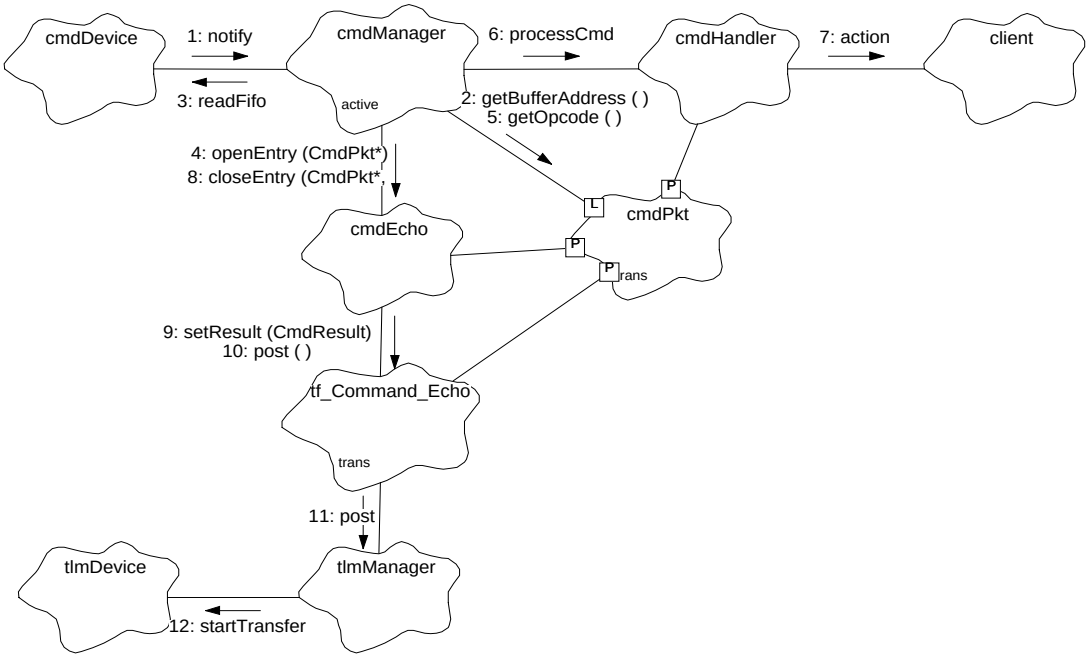
3.8.1 Command Reception and Execution

This section provides a simplified picture of how commands are received, executed, and responded to by the instrument software. For a detailed description of command reception and execution, see Section 16.0 and Section 17.0 .

In order to simplify the command system, the ACIS software is designed to handle one command at a time. As a result, commands must be received and executed as fast as they arrive at the instrument. ACIS is designed to handle no more than 4 commands per second. All commands must be processed in under 250ms.

The following object diagram shows the participants in handling a command, and a simplified sequence of actions which occur when the software processes a command. The numbered actions describe the main steps involved in processing commands sent to the instrument software.

FIGURE 16. Simplified Command Processing Object Diagram



1. A command packet is received by the Back End's RCTU interface hardware, which stores the packet words into a hardware FIFO. Once the entire packet has been received, the hardware generates an interrupt. The interrupt controller dispatches control to the *cmdDevice* object, which notifies the *cmdManager* task object that a command is ready.

2. The *cmdManager* establishes a transient *cmdPkt* object, used to hold the contents of the command packet, and obtains the address of the packet's buffer using getBufferAddress().
3. The *cmdManager* then copies the command packet data from the FIFO into *cmdPkt*'s data buffer, using the *cmdDevice*'s function, readFifo().
4. The *cmdManager* prepares for a command echo using *cmdEcho*.openEntry().
5. The *cmdManager* obtains the command opcode from the packet using *cmdPkt*.getOpcode().
6. The *cmdManager* uses the opcode to select the appropriate command handler object, in this case *cmdHandler*, and tells the selected handler to process the command using processCmd().
7. The handler then performs the required action, usually forwarding action requests onto one or more *client* objects. Once the action has been performed or forwarded, the *cmdHandler* returns a result code (usually provided by the *client*) back to the *cmdManager*.
8. The *cmdManager* passes the result to the *cmdEcho*.closeEntry() to indicate the disposition of the command.
9. The *cmdEcho* then passes the *cmdPkt* and the result code to a command echo telemetry formatter object, *tf_Command_Echo*, which stores a copy of the command and the result code in its telemetry buffer.
10. The *cmdEcho* then tells the formatter to post its telemetry buffer to be sent out of the instrument, using *tf_Command_Echo*.post()
11. The *tf_Command_Echo* object then passes its buffer to the *tlmManager* object, which queues the buffer for transfer out of the instrument.
12. Eventually, the *tlmManager* instructs the *tlmDevice* object to transfer the buffer's contents to the RCTU telemetry interface hardware.

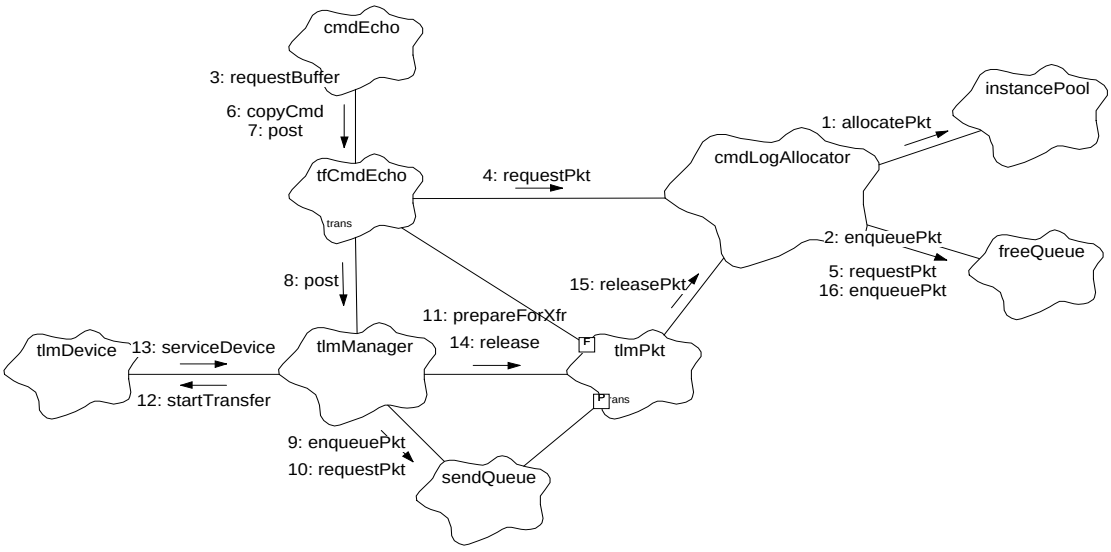
3.8.2 Telemetry Production

This section provides a simplified picture of how telemetry buffers are allocated, formatted, and sent out of the instrument. For a detailed description of telemetry management, buffer allocation and formatting, see Section 18.0 and Section 19.0 .

The ACIS software is required to produce many different types of telemetry item, at different rates, and merge these items into single telemetry stream, at either 24Kbps or 500bps. It is a goal of the ACIS software to avoid gaps in the telemetry stream whenever there is something to send. Since the RCTU transfers words at a rate of 128Kbps, and the telemetry hardware has 64 bits of buffering between the transfer hardware and the RCTU, the software must be capable of starting a new telemetry transfer within 0.5ms, in order to avoid padding between transferred packets.

The following object diagram shows an example of telemetry production and a simplified sequence of actions which occur when the software processes a single telemetry item. The numbered actions describe the main steps involved in producing a telemetry item. This particular scenario uses the *cmdLog* object, described above, as the starting point for illustrating the behavior of the telemetry system.

FIGURE 17. Simplified Telemetry Production Object Diagram



1. During system initialization, each telemetry buffer allocator object, including *cmdLogAllocator*, initializes its respective pools of telemetry packet buffers and free queues. Each then proceeds to allocate every packet in its pool, *instancePool*. Buffer pools are used instead of fixed size arrays of telemetry packet buffers in order to allow easier patching of the number and size of different types of telemetry packets. NOTE: In order to keep different parts of the system reasonably de-coupled, they use different allocators. This way, if one part of the system consumes all of its telemetry

buffers, another part may still be able to send data. If a single allocator were used, then any one part of the system could repeatedly consume all available telemetry buffers, denying the rest of the system the ability to send information.

2. Continuing their initialization, the allocator objects add pointers to the allocated buffers into their respective *freeQueue*'s. Once all buffers have been allocated from their pools and placed into the appropriate *freeQueue*, the buffer initialization is complete.
3. Later, in response to an incoming command, the *cmdEcho* is told to open a log entry by the *cmdManager* (see Figure 16). As part of its processing, the *cmdLog* declares a *tfCmdEcho* object, and tells the object to obtain a telemetry packet buffer.
4. The *tfCmdEcho* object asks *cmdLogAllocator*, the telemetry buffer allocator object used to provide command log and echo telemetry buffers, for a telemetry packet buffer.
5. *cmdLogAllocator* goes to its *freeQueue* and removes a packet buffer from the queue.
6. Once *tfCmdEcho* gets a buffer, *tlmPkt*, from the allocator, the *cmdLog* tells *tfCmdEcho* to copy the command packet contents into the telemetry buffer and to fill in the command result field (not shown).
7. *cmdLog* then tells *tfCmdEcho* to post its buffer to the telemetry manager.
8. *tfCmdEcho*, in turn, passes *tlmPkt* to the *tlmManager*.
9. *tlmManager* appends a pointer to the packet into its *sendQueue*.
10. Later, once all packets ahead of *tlmPkt* have been transferred out of the instrument, *tlmManager* removes *tlmPkt* from the front of the *sendQueue*.
11. *tlmManager* then prepares *tlmPkt* for transfer out of the instrument, obtaining its raw buffer address and the number of words to transfer from *tlmPkt*.
12. *tlmManager* tells the *tlmDevice* to transfer the buffer out of the instrument. *tlmDevice* then programs the Back End's telemetry interface hardware to supply data from the specified buffer to the RCTU serial telemetry port.
13. Once the requested number of words have been transferred from the BEP to the RCTU interface, the hardware generates a telemetry interrupt. At this point, the software has a maximum of 0.5ms to program the telemetry hardware to handle a new transfer before a fill-pattern byte is written to the RCTU by the hardware. The *tlmDevice*'s interrupt handler calls the *tlmManager* directly to service the device.
14. *tlmManager* tells the *tlmPkt* that it is free to be reused. It then attempts to get the next packet from its *sendQueue* and if another packet buffer is ready to be sent, start the next transfer (not shown).
15. *tlmPkt* then tells its allocator, *cmdLogAllocator*, to release the buffer.
16. *cmdLogAllocator* then places *tlmPkt*'s address back into its *freeQueue*. *tlmPkt* is now ready to be re-used.

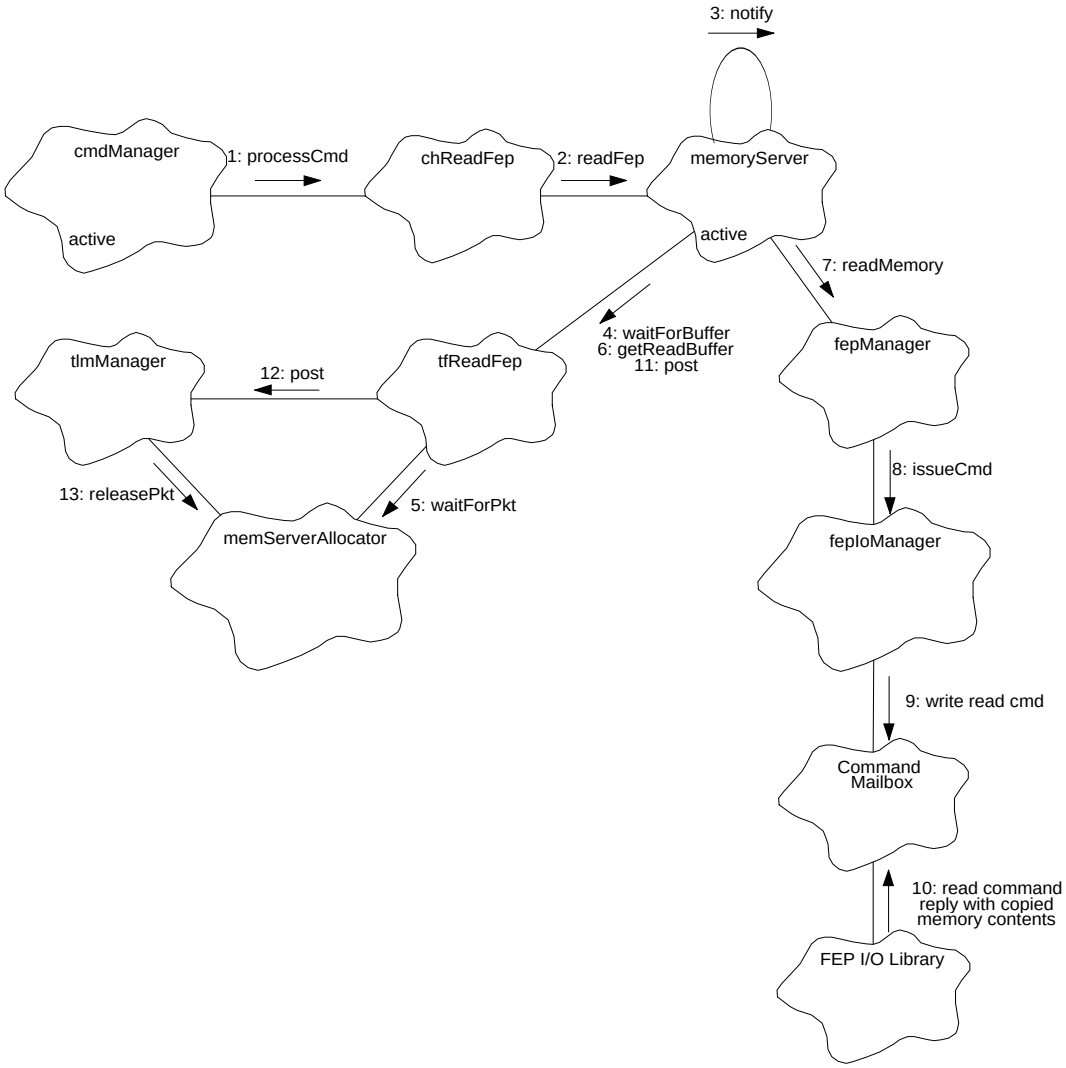
3.8.3 Memory Dumps

This section provides a simplified picture of how requests to dump portions of memory are handled. For a detailed description of memory dump services, and memory load and sub-routine calling services, see Section 27.0 .

All memory functions are handled by a *memoryServer* object. In order to allow for large memory dumps, this object is implemented as a task. This object provides a set of binding functions, which may be safely called from other tasks to request services from the *memoryServer*, and a set of implementation functions, which are used by the *memoryServer*'s task to implement the requested actions.

The following object diagram shows the participants in handling a request to dump the contents of a region of Front End Processor Memory.

FIGURE 18. Simplified Memory Dump Object Diagram



1. The *cmdManager* receives a command packet instructing the system to dump a portion of the memory of one of the Front End Processors. It looks up the corresponding handler object, *chReadFep*, and instructs it to process the command.
2. The *chReadFep* object interprets the contents of the command packet, checking and extracting the index of the FEP from which to read, from which address, and how many words to send. *chReadFep* then tells the *memoryServer* object to perform the read using a binding function provided by the *memoryServer*.
3. The *memoryServer*'s binding function then notifies the task that a request has been registered. Once the request has been registered, the binding function returns to *chReadFep*, which then returns the result of the request to the *cmdManager* object. The *cmdManager* then logs the result with the *cmdEcho*, as shown in Figure 16. Note that the time it takes to execute steps 1 - 3 must be less than 250ms.
4. Later, the *memoryServer*'s task wakes up due to the notification, and proceeds to perform the requested memory dump. It starts by declaring a *tfReadFep* object, which is used to manage and format the telemetry packet buffer containing the dumped data, and telling the object to wait for a telemetry packet buffer to become available.
5. The *tfReadFep* object uses the allocator dedicated to the *memoryServer*, *memServerAllocator*, to attempt to allocate a buffer, or to suspend the task until the *tlmManger* releases one of its packet buffers.
6. Once *tfReadFep* obtains a telemetry packet buffer, the *memoryServer* asks *tfReadFep* to provide the address and maximum length to write the FEP data into.
7. The *memoryServer* task then tells the *fepManager* to read the data from the indicated FEP directly into the buffer address supplied by *tfReadFep*.
8. The *fepManager* forms and issues a request to a given FEP using a corresponding *fepIoManager* object.
9. The *fepIoManager* object writes the read command into the FEP's command mailbox and polls the mailbox for a reply.
10. The software running on the FEP periodically polls its command mailbox. Once it detects that a command is present, it execute the command and writes its reply. In this case, the FEP I/O Library software detects the read request, and copies the requested data into the command mailbox.
11. Once the data request has been satisfied, the *memoryServer* tells the *tfReadFep* object to post its buffer to the telemetry manager.
12. The *tfReadFep* object then passes its telemetry packet buffer to the *tlmManager* object for transfer.
13. Later, once the packet's contents have been transferred out of the instrument, the *tlmManager* object releases the packet's buffer back to the originating allocator, *memServerAllocator*. At this point the buffer is ready to be re-used by the *memoryServer*.

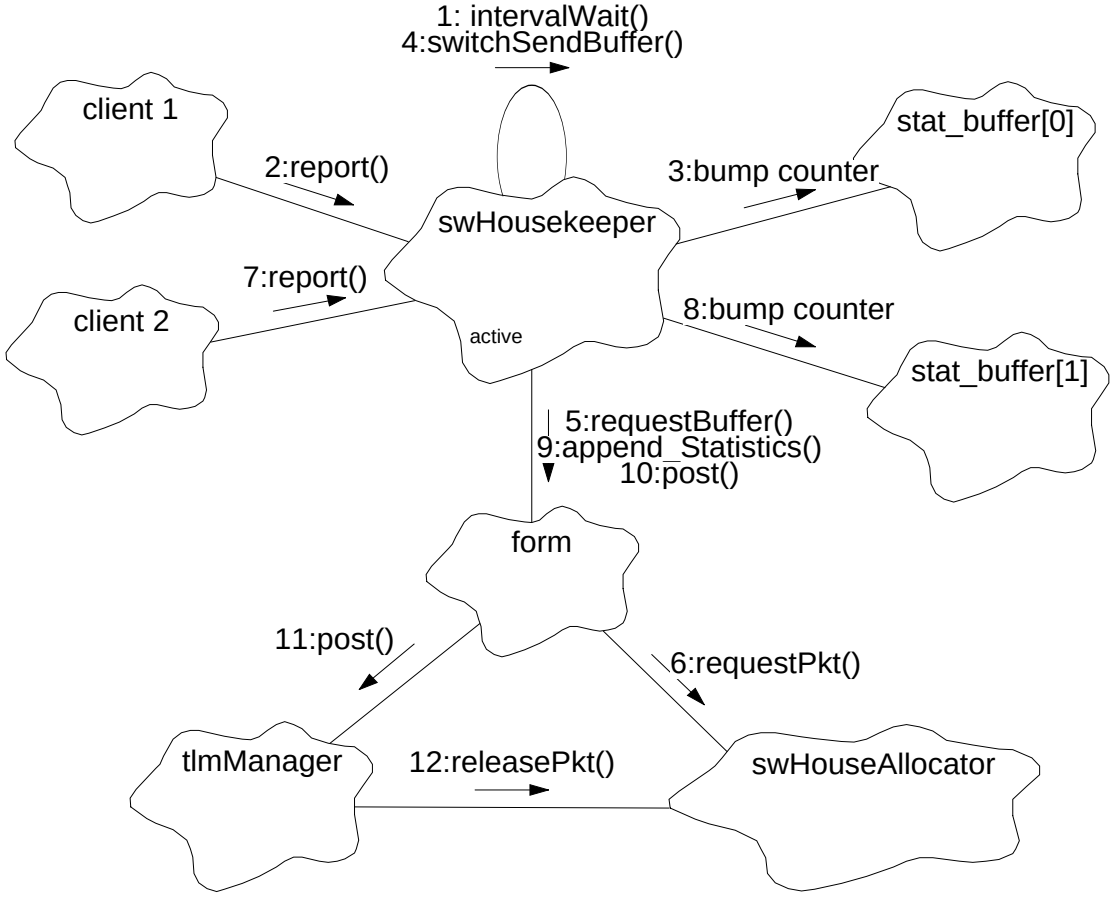
3.8.4 Software Housekeeping

This section provides a simplified picture of how run-time statistics and conditions are acquired and posted to telemetry by the Back End software. For a detailed description of software housekeeping services, see Section 28.0 .

All statistics and warning conditions produced by the Back End Processor software are reported using a *swHousekeeper* object. In order to periodically telemeter the accumulated information, this object is implemented as task. This object provides a binding function, which other tasks use to report information.

The following object diagram shows how various software conditions are reported to housekeeping, and how this information is posted to telemetry.

FIGURE 19. Software Housekeeping Object Diagram



1. During start-up, the *swHousekeeper* object sets up two accumulation buffers, *stat_buffer[0]* and *stat_buffer[1]*, and sets an internal pointer to one of these objects. Later, once multitasking has started, the main loop of the *swHousekeeper* task waits for a period of time, and then telemeters the set of statistic counters accumulated during that period.
2. A client object, *client 1*, running under any thread of control, including interrupts, reports a statistic to the *swHousekeeper* object.
3. The *swHousekeeper* object logs the occurrence into the current accumulation buffer, e.g., *stat_buf[0]*.
4. Periodically, the *swHousekeeper* thread's main loop returns from waiting for the accumulation interval, and proceeds to telemeter the accumulated statistics.
5. The *swHousekeeper* creates a temporary software housekeeping format object, *form*, and tells it to obtain a telemetry packet buffer.
6. The format object, *form*, uses the *swHouseAllocator* object to obtain its telemetry packet buffer. If one is not available at the time of the request, the *swHousekeeper* object reports it to *stat_buffer*[0] and waits for the next cycle. This example, assumes that a packet was obtained.
7. The *swHousekeeper* task then sets its internal pointer to point to *stat_buffer*[1] so that new statistics are recorded into the 2nd buffer while the housekeeper is preparing and telemetering the contents of the first. At this point another client, *client 2*, then may report a statistic.
8. *swHousekeeper* directs its new current telemetry object, *stat_buffer*[1], to accumulate the statistic.
9. *swHousekeeper* copies the contents of the old statistics buffer, *stat_buffer*[0], into the acquired telemetry packet buffer using the format, *form*.
10. Once the information has been copied, the *swHousekeeper* tells the form to posts its telemetry packet buffer for transfer out of the instrument.
11. The *form* object tells the *tlmManager* object to post the packet buffer for transfer.
12. Later, once the packet has been transferred, the *tlmManager* releases the packet buffer back to the originating allocator, *swHouseAllocator*.

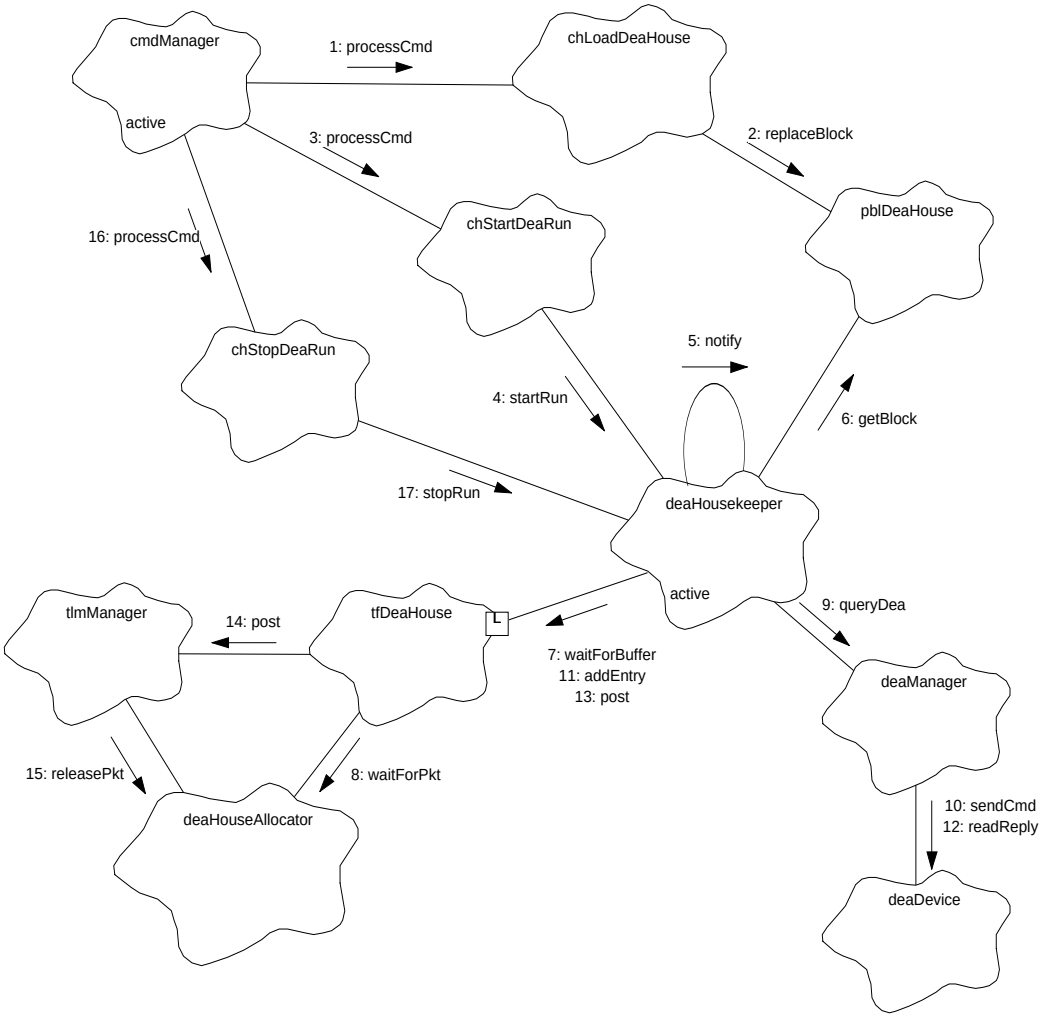
3.8.5 DEA Housekeeping

This section provides a simplified picture of how the instrument acquires housekeeping values from the Detector Electronics Assembly (DEA) and sends them to telemetry. For a detailed description of DEA housekeeping services, see Section 31.0 .

The ACIS software uses the *deaHousekeeper* task to acquire and telemeter housekeeping values from the Detector Electronics Assembly. Like other tasks in the system, the *deaHousekeeper* provides binding functions, which are used by other tasks to command the housekeeper to perform certain services.

The following object diagram shows how various DEA housekeeping classes are configured, and how housekeeping values are acquired and posted to telemetry.

FIGURE 20. DEA Housekeeping Runs Object Diagram



1. The *cmdManager* receives a command packet which contains a DEA housekeeping parameter block to load into the instrument. Using the opcode of the command packet, the cmdManager selects the *chLoadDeaHouse* object to process the command.

2. The *chLoadDeaHouse* object checks the contents of the parameter block, extracts the slot identifier from the block, and then tells the DEA housekeeping parameter block list object, *pblDeaHouse*, to replace the existing parameter block corresponding to the slot with the new parameter block.
3. Later, the *cmdManager* object receives a command packet instructing the instrument to start DEA housekeeping. The *cmdManager* forwards the command to the *chStartDeaRun* object for execution.
4. The *chStartDeaRun* object checks the command, and tells the *deaHousekeeper* object's binding function to instruct the task to start acquiring and sending values.
5. The *deaHousekeeper* object's binding function notifies the task portion of the object to start its operations.
6. Later, as a result of the notification, the *deaHousekeeper* task wakes up, and fetches the parameter block to use from the DEA housekeeping parameter block list, *pblDeaHouse*, and starts its housekeeping operations.
7. Once the parameter block has been fetched, the *deaHousekeeper* object establishes a telemetry object, *tfDeaHouse*, to manage telemetry buffers and formatting. It tells the telemetry object to get a telemetry packet buffer.
8. The *tfDeaHouse* object goes to the housekeeper's buffer allocator, *deaHouseAllocator*, to allocate a telemetry buffer. If one is not immediately available, it waits until a buffer is released by the telemetry manager, *tlmManager*.
9. Once the telemetry object has a buffer, the *deaHousekeeper* tells the *deaManager* to obtain a housekeeping value from the DEA.
10. The *deaManager* object sends a query command to the DEA using the *deaDevice* object, and then waits for and reads the reply to the query, returning the replied value back to the housekeeper.
11. The *deaHousekeeper* then adds the housekeeping value to its telemetry buffer. The *deaHousekeeper* repeats these steps (from step 9) for each housekeeping value specified in the parameter block until all entries have been acquired and stored.
12. Once all of the values have been stored, the *deaHousekeeper* tells *tfDeaHouse* to post its buffer to be sent to telemetry.
13. The *tfDeaHouse* object passes its telemetry buffer to the *tlmManager* object for transmission. The *deaHousekeeper* object then tells the telemetry object to wait for another buffer, and repeats its housekeeping acquisition cycle (from step 8).
14. Later, once the posted telemetry packet buffer has been sent, the *tlmManager* object releases it back to the DEA Housekeeping packet allocator, *deaHouseAllocator*. At this point, the buffer is ready to be re-used by the housekeeper.
15. Finally, the *cmdManager* object receives a command to stop DEA Housekeeping, and forwards the packet to the *chStopDeaRun* object.
16. The *chStopDeaRun* object invokes a binding function of the *deaHousekeeper* object to stop the run. The *deaHousekeeper* then completes its current cycle and stops housekeeping.

3.8.6 Science Runs

This section provides a simplified picture of how the instrument performs science data acquisition and processing runs. For a detailed description of science operations, see Section 33.0 , Section 37.0 , Section 42.0 , Section 43.0 , Section 44.0 and Section 45.0 .

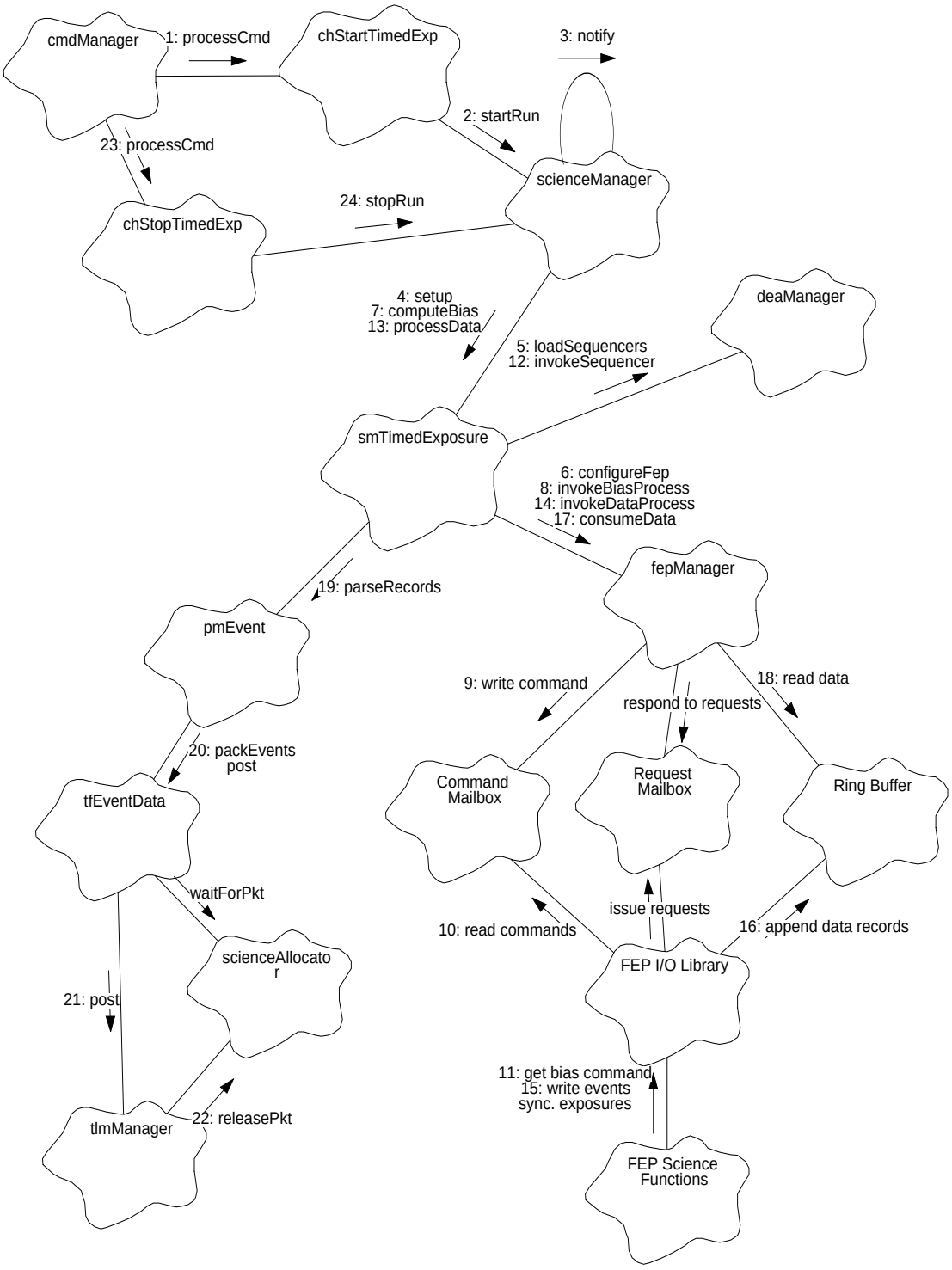
Science processing is managed using a collection of related objects. The *scienceManager* object is a task which is responsible for coordinating the operation of a run. It provides binding functions which are used by other tasks to start and stop science runs.

The *scienceManager* object implements a particular science mode using a particular science mode object. The following example describes a Timed Exposure run, and uses an *smTimedExposure* object to implement the details of the run.

Any given science mode also has various processing modes. These are handled using a processing mode object. For a given run, one processing mode object is used to process the science data from one corresponding Front End Processor being used for the run. The example only illustrates one such object, *pmEvent*.

The following object diagram shows a simplified picture of the overall sequence of events which take place during a science run.

FIGURE 21. Science Run Object Diagram



1. The *cmdManager* object receives a packet instructing the system to start a science run. The *cmdManager* forwards the request to the *chStartTimedExp* object for execution.

2. The *chStartTimedExp* object interprets the command packet, and tells the *scienceManager* object's binding function to start a new science run.
3. The *scienceManager*'s binding function notifies its task to start a new run.
4. Later, as a result of the notification, the *scienceManager*'s task is scheduled to run. It retrieves the parameter block to use (not shown), and tells the science mode object, *smTimedExposure* to setup for a run.
5. The *smTimedExposure* object uses the *deaManager* object to issue commands to the DEA to load the CCD Controller Program and Sequencer RAMs (PRAM, SRAM).
6. The *smTimedExposure* object then tells the *fepManager* object to configure each Front End Processor software.
7. Once the setup is complete the *scienceManager* tells *smTimedExposure* to dump its parameter blocks (not shown). If a bias computation is needed, the manager tells the mode object to compute the pixel-by-pixel bias maps on each of the configured FEPs. If not, skip to step 13.
8. The *smTimedExposure* object tells the *fepManager* object to start the bias computation routines on each of the configured FEPs.
9. For each configured FEP, the *fepManager* object issues a command to the FEP command mailbox to start the bias computation.
10. The FEP I/O Library functions, running on each Front End Processor, read the command.
11. The "start bias" command is then interpreted by the Science functions running on each FEP. At this point, the FEP Science functions proceed to wait for data to arrive from their respective CCDs, and compute the pixel bias map values from the acquired images.
12. Once all of the FEPs are ready to compute their bias levels, and are waiting for images, the mode object, *smTimedExposure*, instructs the *deaManager* object to start the CCD Controller sequencers on the DEA. Once the sequencers start clocking out images, the FEP Science routines proceed to acquire the images, and use the images to build their respective pixel bias maps. Meanwhile, the *smTimedExposure* object periodically polls the FEPs to determine when all of the bias maps are complete. Once the maps are complete, the *smTimedExposure* object then uses the *deaManager* to stop the sequencers.
13. Once the bias maps have been computed, the *scienceManager* object, if configured to do so, notifies the *biasThief* (not shown) to start packing and posting the computed bias maps to telemetry. Once the bias maps have been sent, it instructs the science processing mode object to start acquiring and processing science event data.
14. The *smTimedExposure* object tells the *fepManager* object to start data processing on all of the configured FEPs. Once they're ready, *smTimedExposure* re-starts the DEA sequencers.
15. As the FEP Science functions acquire images and detect events, they use the FEP I/O Library routines to send the data to the Back End Processor.

16. The FEP I/O Library routines append the data and exposure records to the shared-memory ring-buffer.

17. As data is placed into the ring buffer, the *smTimedExposure* object tells the *fepManager* to read the data.

18. The *fepManager* object then reads the exposure and data records from each FEP's ring-buffer.

19. The *smTimedExposure* object uses a collection of processing objects, *pmEvent[]*, to process data. Each *pmEvent* object is associated with one Front End Processor. As data is consumed from a ring-buffer, the *smTimedExposure* object tells the corresponding processing object to process the read exposure and data records.

20. As the *pmEvent* object interprets the records and filters the events, it uses a science data telemetry object, *tfEventData*, to pack the filtered events into a telemetry buffer. If the telemetry object does not have a buffer, *pmEvent* instructs it to wait for a telemetry packet buffer. The *tfEventData* object uses the allocator object dedicated to science processing, *scienceAllocator*, to wait for and allocate the buffer. Once it has a buffer, the event data is packed into the buffer by *tfEventData*. Once the *tfEventData*'s buffer is full, the *pmEvent* object tells the telemetry object to post its buffer for transfer out of the instrument.

21. The *tfEventData* object passes its telemetry packet buffer onto the *tlmManager* object to be posted for transfer out of the instrument.

22. Later, once the telemetry packet buffer contents have been transferred, the *tlmManager* releases the packet buffer back to the allocator, *scienceAllocator*. At this point, the telemetry packet buffer can be re-used by science.

23. Eventually, the *cmdManager* receives a command instructing the system to stop performing science. The *cmdManager* forwards the packet to the *chStopTimedExp* object for execution.

24. The *chStopTimedExp* object then instructs the *scienceManager*'s binding function to stop the run. The *scienceManager*'s binding function tells *smTimedExposure* to stop, which then sets an internal flag and notifies the task to stop. When the task detects the stop request, it instructs the FEPs to finish processing. Once the last exposure being processed by the Front End Processors is complete, *smTimedExposure* stops the DEA CCD Controller sequencers, and posts a summary of the run to telemetry (not shown).

4.0 Interfaces (36-53204 02-)

This section describes the main interfaces used by the ACIS Science Instrument Software.

4.1 R3000 Core Processor Interfaces

The Mongoose Microcontroller is based on an R3000 processor. The following sections provide a brief overview the R3000 Core Processor. This information is taken from “MIPS RISC Architecture,” by Kane, published by Prentice Hall, and from “The MIPS Programmer’s Handbook,” by Farquhar and Bunce, published by Morgan Kaufmann. A description of the Mongoose extensions to the R3000 is provided in Section 4.2 .

4.1.1 Instruction Set

This section provides a brief overview of some aspects of the R3000 instruction set. The R3000 is a reduced-instruction-set-computer (RISC) and has a reasonably small instruction set. Each processor instruction occupies 1 32-bit word, and, barring memory waits, is effectively executed in 1 machine cycle. Certain instructions, however, require the compiler, assembler or software engineer to be careful in the ordering of instructions. For example, memory load instructions prohibit the instruction immediately following it from using the fetched value, and conditional and unconditional branch instructions always execute the instruction immediately following the branch instruction, regardless of whether the branch is taken or not.

The R3000 assemblers are “smart” in that they provide enhanced instructions, and make code-generation decisions when possible. For example, most of the assemblers provide a “Load Constant Address” instruction which the assembler may translate into one or two core instructions depending on the nature of the address. If the address has non zero values in both the upper and lower 16-bits, the assembler will produce a “lui” to load the upper 16-bits, followed by a “ori” to add in the lower 16-bit of the address. If the lower 16-bits are zero, the assembler will omit the “ori” operation. If the upper 16-bits are zero, the assembler will just use a “li” instruction to load the lower 16-bits and zero the upper 16.

The following is a list of the core R3000 instructions provided by the processor:

TABLE 4. R3000 Core Instruction Set

Mnemonic	Instruction
	Arithmetic Instructions
add	Add
addi	Add Immediate (16-bit)
addiu	Add Unsigned Immediate
addu	Add Unsigned
and	AND
andi	AND Immediate
div	Divide

TABLE 4. R3000 Core Instruction Set

Mnemonic	Instruction
divu	Divide Unsigned
mult	Multiply
multu	Multiply Unsigned
nor	NOR
or	OR
ori	OR Immediate
sll	Shift Logical Left
sllv	Shift Logical Left Variable
sra	Shift Right Arithmetic
srav	Shift Right Arithmetic Variable
srl	Shift Logical Right
srlv	Shift Logical Right Variable
sub	Subtract
subu	Subtract Unsigned
xor	XOR
xori	XOR Immediate
	Branch and Jump Instructions
bczf	Branch Co-processor Z Flag False
bczt	Branch Co-processor Z Flag True
beq	Branch if Registers Equal
bgez	Branch if register >= 0
bgezal	Branch and Link if register >= 0
bgtz	Branch register > 0
blez	Branch register <= 0
bltz	Branch register < 0
bltzal	Branch and Link register < 0
bne	Branch if registers not equal
j	Jump
jal	Jump and Link
jlr	Jump and Link Register
jr	Jump Register
	Co-processor Instructions
cfcz	Move from Co-processor Z Control Register
copz	Co-processor Z Operation
ctcz	Move to Co-processor Z Control Register
lwcz	Load Word to Co-processor Z Register
mfc0	Move from System Co-processor Register
mfcz	Move from Co-processor Z Register
mtc0	Move to System Co-processor Register
mtcz	Move to Co-processor Z Register
tlbp	System Co-processor Probe TLB for Match

TABLE 4. R3000 Core Instruction Set

Mnemonic	Instruction
tlbr	System Co-processor Read Indexed TLB Entry
tlbwi	System Co-processor Write Indexed TLB Entry
tlbwr	System Co-processor Write Random TLB Entry
rfe	System Co-processor Return from Exception
swcz	Store Word from Co-processor Z Register
	Load and Store Instructions
lb	Load byte
lbu	Load byte unsigned
lh	Load half-word
lhu	Load half-word unsigned
lui	Load Upper 16-bits immediate
lw	Load Word
lwl	Unaligned Load Word Left
lwr	Unaligned Load Word Right
mfhi	Move to High Accumulator (mult,div)
mflo	Move from Low Accumulator
mthi	Move to High Accumulator
mtlo	Move to Low Accumulator
sb	Store Byte
sh	Store Half-word
sw	Store Word
swl	Unaligned Store Word Left
swr	Unaligned Store Word Right
	Miscellaneous Instructions
break	Breakpoint
slt	Set on Less Than
slti	Set on Less Than Immediate
sltiu	Set on Less Than Immediate Unsigned
sltu	Set on Less Than Unsigned
syscall	System Call

4.1.2 System Co-processor (C0)

This section provides an overview of the R3000's System Co-processor (CU0). This co-processor's registers can be read via the "mfc0" CPU instruction, and written to via the "mtc0" CPU instruction. This co-processor provides the following registers:

- Status Register ($12) - This read-write register controls the overall state of the processor mode and interrupt masks.

- Cause Register ($13) -This register supplies interrupt information and controls the state of the software interrupt bits.
- Exception Program Counter ($14) - This register contains the address to return to after processing the current exception or interrupt.
- Bad Virtual Address Register ($8) - This register contains the virtual address which caused the current address error exception.
- Context Register ($4) - The register duplicates the information in the Bad Virtual Address Register, but in a form more appropriate for software processing of the translation look-aside buffer (not present on the Mongoose)
- Processor Revision Identification Register ($15) - This register identifies the processor version. It is not present on the Mongoose, and attempts to read this register return 0.

4.1.2.1 Status Register (C0_SR)

This section describes the R3000 System Co-processor's Status Register. The following lists the bit fields contained within this register:

31 28	27 23	22	21	20	19	18	17	16	15 8	7 6	5	4	3	2	1	0
CU	0	BEV	TS	PE	CM	PZ	SwC	IsC	IntMask	0	KUo	IEo	KUp	IEp	KUc	IEc

IEc, IEp, IEo ..... Interrupt Enable (Current, Previous, Old)
KUc, KUp, KUo ..... Kernel/User Mode (Current, Previous Old)
IntMask ..... Interrupt Mask
IsC ..... Isolate Cache (unused on Mongoose)
SwC ..... Swap Caches (unused on Mongoose)
PZ ..... Parity Zero (unused on Mongoose)
CM ..... Cache Miss (unused on Mongoose)
PE ..... Parity Error (unused on Mongoose)
TS ..... TLB Shutdown (unused on Mongoose)
BEV ..... Bootstrap-Exception Vectors
CU (0-3) ..... Co-processor Usable (CU0 only on Mongoose)

The following is a detailed breakdown of the IntMask and Co-processor Usable bits:

15	14	13	12	11	10	9	8
INT6	INT5	INT4	INT3	INT2	INT1	Sw2	Sw1

Sw ..... Software Interrupts
INT ..... Hardware Interrupts

31	30	29	28
CU3	CU2	CU1	CU0

CU0 ..... System Co-processor Usable (must always be 1)
CU1 ..... Math Co-processor Usable (on ACIS, always 0)
CU2 ..... Co-processor 2 Usable (on ACIS, always 0)
CU3 ..... Co-processor 3 Usable (on ACIS, always 0)

Interrupt Enable Bits.....These bits define whether interrupts are enabled or not (0=disabled, 1=enabled). The IEc bit is cleared upon reset or hardware interrupt. The software uses this bit to disable all interrupts, and to enable those interrupts indicated by the IntMask field. When an interrupt or exception occurs, the IEc bit is copied in to the IEp slot, and the IEp slot is copied into the IEo slot. When a return-from-exception co-processor instruction (rfe) is executed, the previous contents of the IEp and IEc bits are restored. The IEo bit is left unmodified.

Kernel/User Mode Bits .....These bits define whether the processor is in Kernel or User mode (0=kernel mode, 1=user mode). The ACIS software always runs in Kernel mode.

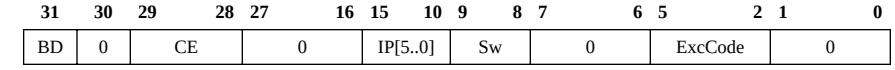
Interrupt Mask.....These bits are used by the software to individually enable or disable interrupts tied to the corresponding mask bits (0=disable, 1=enable). The Software Interrupts (Sw1, Sw2) are caused by writing the associated bits in the Cause Register. The remaining bits (INT1..INT6) are cause by physical hardware devices connected to the corresponding interrupt lines. The Mongoose Processor reserves INT3 for the extended Mongoose interrupts and exceptions.

Bootstrap Exception Vectors.....This bit is set by hardware upon reset, and may be set or cleared by software. When set, this bit causes the General Exception Vector to be mapped into the Boot ROM address space (virtual address 0xbfc000c0). When this bit is 0, the General Exception Vector is mapped into Instruction Cache space (virtual address 0x80080080).

Co-processor Usable Bits.....These bits indicate which R3000 System Co-processor s are present and usable (0=unuseable, 1=usable). On ACIS, only the System Co-processor (which contains this Status Register) is usable.

4.1.2.2 Cause Register (C0_CAUSE)

This section describes the R3000 System Co-processor's Interrupt Cause Register.



ExcCode ..... Exception Code
Sw ..... Software Interrupts
IP[5..0] ..... Interrupt Pending
CE ..... Co-processor Error
BD ..... Branch Delay

ExcCode.....This field contains the Exception Code for the current exception. It may contain one of the following values:

TABLE 5. Cause Register Exception Code Values

Code	Cause
0 - Int:	External Interrupt (including Mongoose CSI)
1 - MOD:	TLB Modification exception (not on Mongoose)
2 - TLBL:	Load or fetch TLB miss exception (not on Mongoose)
3 - TLBS:	Store TLB miss exception (not on Mongoose)
4 - AdEL:	Load or fetch Address Error Exception
5 - AdES:	Store Address Error Exception
6 - IBE:	Instruction Bus Exception
7 - DBE:	Data Bus Exception
8 - Sys:	SysCall Exception
9 - Bp:	Breakpoint Exception
10 - RI:	Reserved Instruction Exception
11 - CpU:	Co-processor Unusable Exception
12 - Ovf:	Arithmetic Overflow Exception
13-15:	Reserved

Sw[0,1].....These bits correspond to the pending software interrupts. Writing to these bits sets or clears the software interrupts

IP[0..5] .....These bits correspond to the pending external interrupts.

CE[0..1].....These bits identify which co-processor cause a Co-processor Unusable Exception

BD ..... This bit indicates whether an exception was taken while executing in a branch delay slot (i.e. the instruction immediately following a branch).

4.1.2.3 Exception Program Counter (C0_EPC)

This section describes the R3000 System Co-processor's Exception Program Counter Register. This register contains the address to resume after processing an interrupt or exception. Typical code for ending an exception is as follows:

```
mfc0   k0,C0_EPC  # Get return address into kernel register 0
nop               # 1 instruction delay after a mfc0 before value valid
jr  k0            # Jump to resumption address
rfe               # Use delay slot to tell CU0 to restore flags
```

4.1.2.4 Bad Virtual Address Register (C0_BVADDR)

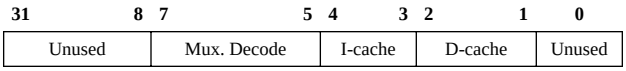
This section describes the R3000 System Co-processor's Bad Virtual Address Register. This register contains the virtual address which caused an address exception. This register is not used for bus errors.

4.2 Mongoose Hardware Interface

The following sections provide a brief overview of the Mongoose Microcontroller registers. This information is taken from “Mongoose ASIC Microcontroller Programming Guide, September 1993,” by Brian Smith, NASA Reference Publication 1319.

4.2.1 Microboot Control Word (0xbffffffc)

This section describes the Mongoose Microboot Control Word. This word is located in the last location in Boot ROM (or RAM in the case of the FEPs) and is used by the Mongoose during hardware initialization (prior to any software activity) to configure its Instruction and Data cache memory chip layout. The format of this register is as follows:



D-cache .....Number of D-cache Chip Selects [0:One chip select, 1:Two chip selects, 2:Illegal, 3:Four chip selects]

I-cache .....Number of I-cache Chip Selects [0:One chip select, 1:Two chip selects, 2:Illegal, 3:Four chip selects]

Mux. Decode .....Cache chip select address mux. decode [0:decode address(13:12), 1:decode address(14:13), 2:decode address(15:14), 3:decode address(16:15), 4:decode address(17:16), 5:decode address(18:17), 6:decode address(19:18), 7:all chip selects not active]

For the Back End Processor, the value to use for this control word is 0x0ba.

For all of the Front End Processors, the value to use for this control word is 0x0a0.

4.2.2 Command/Status Interface (CSI) Registers (0xb4c00000)

This section provides an overview of the Mongoose Command and Status interface registers. The Mongoose Microcontroller provides a block of command and status interface registers, located at address 0xb4c00000. This block contains the following registers:

- Configuration Register - This register enables/disables the general purpose timer, controls the current DMA transfer mode, and establishes the number of wait states used for uncached memory addresses.
- Instruction Cache Address Register - This register is used to command a read or write from Instruction Cache RAM.
- Instruction Cache Data Register - This register is used to transfer data words to and from Instruction Cache RAM.
- Extended Interrupt Mask Register - This register selectively enables/disables interrupts from the various Mongoose devices.
- Extended Interrupt Cause Register - This register indicates individual pending interrupts from the Mongoose devices and is used to reset the pending interrupt.
- DMA Controller Origin Address - This register is used to set the starting source address of a DMA transfer.

- DMA Controller Destination Address - This register is used to set the starting destination address of a DMA transfer.
- DMA Controller Count - This register is used to control the number of words to copy in a DMA transfer.
- Timer Count - This register contains and controls the current general purpose timer count. When the general purpose timer is enabled, this counter decrements with each CPU cycle. When the count reaches 0, a timer interrupt is generated.
- Watchdog Timer Count - This register contains and controls the current Watchdog timer count. When hardware has enabled the counter, this counter decrements with each CPU cycle. When the count reaches 0, an external reset pulse is generated.
- Uart Command/Status - When written to, this register configures the Mongoose Uart. When read, this register provides status information about the Uart.
- Uart Data - This register is used to write a byte of data to transmit out the Mongoose Uart's serial port, and is used to read a byte of data received by the Uart.

The following illustrates the memory layout for this block of registers:

Extended Interrupt Mask.....0xbf400000

Extended Interrupt Cause.....0xbf400004

Configuration Register 0xbf400008

DMA Origin.....0xbf40000c

DMA Destination.....0xbf400010

DMA Word Count.....0xbf400014

Watchdog Count.....0xbf400018

Timer Count 0xbf40001c

Uart Command/Status.....0xbf400024

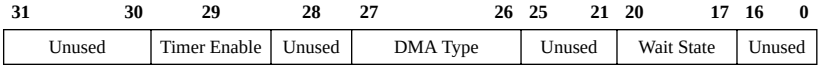
Uart Data.....0xbf400028

I-cache Address.....0xbf40002c

I-cache Data 0xbf400030

4.2.3 Configuration Register

This section describes the Mongoose Configuration Register. This register is used to enable/disable the general purpose timer, to initiate memory transfers using the DMA Controller, and to select the number of memory wait-states to use in external, uncached memory. The format of the configuration register is as follows:



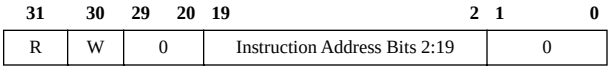
Wait State .....Number of memory wait-state cycles to use [0:15]

DMA Type.....DMA Transfer Mode [0:Off, 1:D-Cache to Memory, 2:Memory to D-Cache, 3:Memory-to-Memory]

Timer Enable .....General Purpose Timer Enable/Disable Control [0:Disabled, 1:Enabled]

4.2.4 Instruction Cache Access

This section describes the Mongoose's Instruction Cache access registers and the procedure used to read and write the contents of the Instruction Cache. The Mongoose Instruction cache is loaded and dumped using an address and data register. The format of the Mongoose Instruction Cache Address register is as follows:



Instruction Address..... Source or destination word address within I-cache

W ..... Write-to-Icache Control Bit

R ..... Read-from-Icache Control Bit

To write into the instruction cache, clear the bits 0, 1 and 20 through 31 of the destination address, OR a 1 into the Write-to-Icache control bit, write the desired data word to the data register, wait at least one CPU cycle, then write the formatted address to the Instruction Cache address register.

To read a word, zero bits 0, 1 and 20 through 31 of the source address, OR a 1 into the Read-command bit, write the formatted address into the Instruction Cache address register, wait at least one CPU cycle, and then read the Instruction Cache data register.

4.2.5 Extended Interrupts

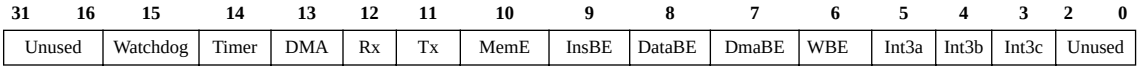
This section describes the Mongoose Extended Interrupts. The Mongoose supports a set of extended interrupts and exceptions, which when asserted and unmasked, cause INT3 (IP[2]) to be asserted on the R3000 Core (see Section 4.1.2.1 and Section 4.1.2.2). The set of interrupts and exceptions include:

- Watchdog Timer End of Count Interrupt
- General Purpose Timer End of Count Interrupt
- DMA Done Interrupt
- UART RX Ready Interrupt
- UART TX Ready Interrupt

- External Memory Bus Time-out Exception
- Instruction Read Bus Error Exception
- Data Read Bus Error Exception
- DMA Bus Error Interrupt
- Write Bus Error Interrupt
- External Interrupt 3a
- External Interrupt 3b
- External Interrupt 3c

4.2.5.1 Extended Mask Register

This section describes the Mongoose Extended Interrupt Mask Register. This register is used to prevent one or more of the extended interrupt sources from causing an interrupt. A ‘1’ in the mask allows interrupts from the corresponding source, and a ‘0’ prevents the source from interrupting the processor. The format of the extended mask register is as follows:



Int3c ..... External Interrupt 3c
Int3b ..... External Interrupt 3b
Int3a ..... External Interrupt 3a
WBE ..... Write Bus Error
DmaBE ..... DMA Bus Error
DataBE ..... Data Read Bus Error
InsBE ..... Instruction Read Bus Error
MemE ..... External Memory Bus Time-out
Tx ..... UART Transmit Ready Interrupt
Rx ..... UART Receiver Ready Interrupt
DMA ..... DMA Transfer Complete Interrupt
Timer ..... General Purpose Timer End of Count Interrupt
Watchdog ..... Watchdog TImer End of Count Interrupt

4.2.5.2 Extended Cause Register

This section describes the Mongoose Extended Cause Register. When read, this register returns the list of sources which have requested interrupt service. A ‘1’ indicates that the corresponding device has asserted an interrupt request. A ‘0’ indicates that no interrupt request is pending from the device. When written to, this register clears the pending request. The format for this register is identical to that for the Extended Mask Register (see Section 4.2.5.1).

4.2.6 DMA Controller

This section describes the Mongoose DMA Controller. The Mongoose DMA Controller is configured using the DMA Controller Starting Origin Register, DMA Controller Starting Destination Register, the DMA Controller Count Register, and the DMA Type bits in the Mongoose Configuration Register (see Section 4.2.3).

- To disable the DMA, write 0s to bits 26 and 27 in the Configuration Register
- To copy data from one region of memory to another, disable the DMA, store the starting source and destination addresses in the Origin and Destination Registers, respectively, set the Count register to the number of 32-bit words to copy, the write two 1s to bits 26 and 27 of the Configuration Register to start the transfer. Once the transfer completes, the DMA Done bit in the Extended Interrupt Cause register will be asserted, and if enabled, a DMA Done interrupt is generated.
- To copy data from Memory to Data Cache, the steps are the same as above, except that a 0 and 1 are written to bits 26 and 27 of the Configuration Register, respectively.
- To copy data from Data Cache to Memory, the steps are the same as above, except that a 1 and 0 are written to bits 26 and 27 of the Configuration Register, respectively.

4.2.7 Watchdog Timer

This section describes the Mongoose Watchdog Timer. The Watchdog Timer’s time-out duration is established by writing a value into its Count Register. Upon reset, the counter’s value is 0xffffffff. This count register decrements with each CPU cycle (10MHz). When the count reaches 0, the timer generates a hardware pulse, which on ACIS, is connected to the processor’s reset logic.

4.2.8 General Purpose Timer

This section describes the Mongoose General Purpose Timer. The General Purpose Timer’s time-out duration is established by writing a value into its Count Register. Upon reset, the counter’s value is 0xffffffff, and the timer is disabled. When enabled via the Configuration Register (see Section 4.2.3), this count register decrements with each CPU cycle (10MHz). When the count reaches 0, the timer asserts the Timer End of Count extended interrupt (see Section 4.2.5), and the counter wraps to 0xffffffff and continues to decrement with each CPU cycle.

4.3 Back End Processor Hardware Interfaces

The following sections describe the Back End Processor hardware interfaces. This information is taken from the “DPA Hardware Specification and System Description, Rev. 04,” by Dorothy Gordon, MIT Drawing Number 36-02104. Additional information is included in advance of future publications of this document, and is marked as TBD.

4.3.1 Memory Map Overview

This section illustrates the overall Back End Processor hardware memory map:

TABLE 6. Back End Processor Memory Map Overview

Virtual Address	Byte Size	Name	Use
0xbfc00000	0x8000	Boot ROM	Contains Bootstrap loader software (see Section 13.0) and Microboot Control Word (see Section 4.2.1)
0xbf800000	N/A	External Debug Board	Debug board used for development/pre-flight testing
0xbf400000	0x30	Mongoose CSI Subblock	see Section 4.2
0xb8000000	0x100000	Flight ROM	Contains the delivered image of the instrument software code and initialized data. This image is copied into I-cache and D-cache by the Bootstrap loader software.
0xa8000000	0x7ff0000	FEP Shared Memory Addresses	Shared FEP RAM used for mailboxes, ring-buffers, and bias maps and bias parity planes.
0xa0180000	N/A	BEP I/O Devices	see below
0xa0000000	1 MB	Bulk Memory and FEP Shared Memory Addresses	BEP-populated RAM used for telemetry buffers.
0x80080000	0x80000	Instruction Cache (SEU-immune)	Contains copy of patched and running code. Since this RAM can only be written to via the Mongoose CSI (see Section 4.2.4), portions of this area are used to preserve patch lists, system configuration parameters, science parameter blocks, and DEA parameter blocks across resets.
0x80000000	0x20000	Data Cache (SEU-immune)	Contains copy of patched initialized data. Used for run-time stacks, RTX resources, and program data.

4.3.2 Control Register (0xA0180000)

This section describes the Back End Processor's Control Register. This read/write register controls the state of various Back End Processor devices. The format of this register is as follows:

31 12	11 6	5 2	1	0
Unused	FEPPOW[0-5]	SW_BLT0 - SW_BLT3	DNLKTRENB	UPLKENB

UPLNKENB ..... Enable command data reception
DNLKTRENB..... Enable telemetry data transmission
SW_BLT(0-3)..... BiLevel discretes
FEPPOW[0-5] ..... FEP power control

4.3.3 Status Register (0xA0180004)

This section describes the Back End Processor's Status Register. This read-only register provides status information on the various Back End Processor Devices. A '1' in a given bit indicates the presence of the corresponding condition. A '0' indicates that the corresponding condition is not currently present. Some conditions persist until reset via the Pulse Register (see Section 4.3.4). The format of this register is as follows:

23 22	21	20	19	18	17	16	15	14	13	12 7	6	5	4	3	2	1	0
RS[1-0]	BID	CR	WD	RAD	ST	BT	X	DSR	DCB	FI(0-5)	RM	DI	UFE	UFF	UI	UFA	UE

UE ..... UPLKERR: Error in command data reception
UFA..... UPLKFULLAT: TBD
UI ..... UPLKINT: End of packet or error condition
UFF ..... UPLKFULL: Command FIFO full
UFE..... UPLKFEMP: Command FIFO empty
DI ..... DNLKINT: Downlink transfer complete
RM ..... RADMONDET: Spacecraft has detected high radiation
FI(0-5) ..... FEPINT0-5: FEP interrupt requested
DCB..... DEACMDBUSY: DEA receiving data
DSR..... DEASTATRDY: DEA has sent status data
X ..... unused
BT ..... BOOT_MOD: Boot modifier bit asserted
ST ..... STAN_MOD: Hardware in standby mode
RAD ..... RAD_MON: High radiation environment detected
WD ..... WDOGRST: Watchdog reset occured
CR ..... CMDRST: Command reset issued
BID ..... BEP ID set via backplane slot
RS[1-0]..... ROM_SEG[1-0]: Segment of Boot Rom used for boot.

4.3.4 Pulse Register (0xA0180008)

This section describes the Back End Processor's Pulse Register. This write-only register resets various status and interrupt conditions. Writing a '1' to a given bit performs the corresponding action. Writing a '0' to a bit has no effect on the corresponding item. The format of this register is as follows:

13	12	11	10	9	8 3	2	1	0
FIFORST	DEARST	CLRCMD	CLRWD	CLRDEA	CLRFEPINT(0-5)	CLRRMD	CLRDI	CLRUI

CLRUI ..... CLRUPLKINT:Clear Uplink Controller interrupt
CLRDI ..... CLRDNLKINT:Clear Downlink Transfer Controller interrupt
CRLRMD ..... CLRRADMONDET:Clear Radiation Monitor Detect interrupt
CLRFEPINT(0-5) ..... CLRFEPINT0-5:Clear Front End Processor interrupt (1 bit for each FEP)
CLRDEA ..... CLRDEASTATRDY:Clear DEA Status Port Ready flag
CLRWD ..... CLRWDOGRST:Clear Watchdog Reset flag
CLRCMD ..... CLRCMDRST:Clear Commanded Reset flag
DEARST ..... DEARST:Reset Detector Electronics Assembly
FIFORST ..... FIFORST:Reset Command FIFO

4.3.5 Command FIFO and Uplink Controller (0xA0180014)

This section describes the Back End Processor's Command FIFO. As the Back End Processor hardware receives 16-bit command words from the Remote Command and Telemetry Unit (RCTU), it writes these words into a 256-word deep hardware FIFO. The hardware always stores received words into the FIFO, even when the Uplink Controller is disabled.

When enabled, the Uplink Controller monitors the acquisition of ACIS packets. After being enabled, the controller interprets the first received 16-bit word as the total number of words in the incoming ACIS command packet. It then counts down each received word until all the words of the packet have been acquired. It then generates an interrupt to the Back End Processor, indicating that a complete packet has been received. It then interprets the next 16-bit word as the word count of the next command packet. The format of this register is as follows:

31	**30**	**17**	**16**	**15**	**0**
FNE	Undefined		PFW	Command Word	

Command Word ..... 16-bit command word
PFW ..... Packet First Word Flag
FNE ..... FIFO-Not-Empty

4.3.6 Downlink Transfer Controller

This section describes the behavior of the Back End Processor's Downlink Transfer Controller (DTC). This controller is responsible for sending telemetry packets from bulk, uncached memory, to the Remote Command and Telemetry Unit (RCTU) interface hardware. The software initiates a packet transfer by programming the start address and end address of the block of words to telemeter, and then enabling the controller via the Back End's Control Register (see Section 4.3.2). Once enabled, the controller autonomously copies words from the bulk memory to the RCTU interface as the spacecraft demands telemetry data from the serial data port. Once the last word has been copied from bulk to the controller's RCTU interface logic, the DTC causes an interrupt to indicate that a new transfer may be started.

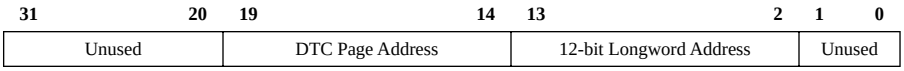
The DTC provides effectively a 64-bits FIFO between bulk memory and the RCTU. If no data is available to send when the spacecraft asks for data, the DTC supplies a fill pattern, 0xb7

The RCTU will request data no faster than 128Kbps. To avoid the fill pattern when there is data to send, the software has 0.5ms after being interrupted to start a new transfer.

The transfer controller uses a page-based addressing scheme, where each page contains 4096 continguous 32-bit words. Each programmed transfer must be within a 4096 word page block, and must not cross a page boundary.

4.3.6.1 DTC Start Address Register (0xA0180018)

This register specifies the starting location of a block of data to be telemetered by the Downlink Transfer Controller. The addresses stored in this register must be within bulk (uncached) memory, and must be on a 32-bit boundary. The following illustrates the format of the start address register:



12-bit Longword Address..... This is the offset, relative to the start of the page, of the first 32-bit word to transfer.

DTC Page Address ..... This is the page containing the data to transfer

To set the starting address of the transfer, take the virtual address of the buffer in bulk memory, and mask off the lower two bits and upper 12 bits of the address and write the result in to the DTC Start Address Register.

4.3.6.2 DTC End Address Register (0xA018001C)

This register specifies the last location of a block of data to be telemetered by the Downlink Transfer Controller. The addresses stored in this register must be within bulk (uncached) memory, must be on a 32-bit boundary, and must be greater than the current address stored in the DTC Start Address Register. The format of this register is identical to the DTC Start Address Register, except that the DTC Page Address bits are unused.

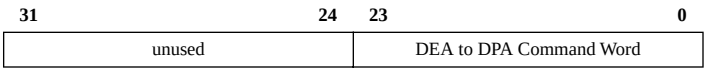
4.3.6.3 DTC Address Count Register (0xA0180020)

This register specifies the current longword count of a DTC transfer in progress. Bits 0 and 1 always read back as zero.

4.3.7 DEA Command Register (0xA0180010)

This section describes the Back End Processor’s Detector Electronics Assembly (DEA) Command Register. This register is used to send command words to one or more of the DEA subsystems. When a word is written into this register, the hardware sets the DEA Command Port busy flag in the Back End’s Status Register (see Section 4.3.3), and serially shifts the command word to the DEA. Once the entire command has been sent to the

DEA, the hardware de-asserts the busy flag in the status register, indicating that the port is ready for the next command.



4.3.8 DEA Status Register (0xA018000C)

This section describes the Back End Processor's Detector Electronics Assembly (DEA) Status Register. This register is used to hold 24-bit status words sent by one of the DEA to the Back End Processor in response to a query command. Once a status word has been received, the hardware sets the DEA Status Port has data flag in the Back End Processor's Status Register (see Section 4.3.3). This flag remains asserted until cleared by writing a '1' to the "Clear DEA Status Port ready flag" in the Pulse Register (see Section 4.3.8).

4.3.9 S/C Counter - Latched Count (0xA0180024)

This register contains a 32-bit value of the Spacecraft counter. This value indicates the time of the last command sent to the DEA..

4.3.10 S/C Counter - Running Count(0xA0180028)

This register contains a 32-bit value of the running Spacecraft counter.

4.3.11 FEP Shared Memory Interface (0xA8000000)

This section describes the Back End Processor's interface to each Front End Processor's shared memory section. Each Front End Processor contains several sections of memory, including the Front End Processor's Boot RAM, which are shared with the Back End Processor. These memory sections are mapped into the Back End Processor's virtual address space, and can be read from and written to simultaneously by Front End Processor and the Back End Processor. The shared memory interfaces are used by the Back End Processor to perform the following functions:

- Load a FEPs Microboot Control Word into its Boot RAM (only while the FEP is reset)
- Load bootstrap code into a FEPs Boot RAM (only while the FEP is reset)
- Issue commands to a FEP via a shared-memory BEP-to-FEP command mailbox
- Receive requests from a FEP via a shared-memory FEP-to-BEP request mailbox
- Load bad pixel and column codes directly into the FEPs Pixel Bias Map
- Directly read the contents of a FEPs Pixel Bias Map

The base address for the FEP shared memory is listed in the memory map (and above). To access the shared memory for a particulare FEP, the fep id (0x00 - 0x05) is added to the upper 16 bits of the base address.

4.3.12 BEP to FEP Command Registers

The BEP to FEP Command registers physically reside in each Front End Processor, but are controlled by the Back End Processor via the shared memory interface. See section Section 4.4.10 for a detailed explanation of these registers.

4.3.13 Interrupts

This section describes the Back End Processor interrupts. The following table lists the interrupts used by the Back End Processor, describes the priorities, interrupt input assignments and timing requirements of each interrupt:

TABLE 7. Back End Processor Interrupts

Priority (0 highest)	Input	Name	Causes	Response Time	Frequency
0	INT0	Command Interrupt	Command Ready, Length Error, FIFO Full	< 1ms	peak 4/second avg. 1/hour
1	INT1	DTC Interrupt	Telemetry transfer complete	Goal < 0.5ms	peak 100/second avg. 3/second
2	INT3[a]	Timer Tick	Timer Count expired	< 1ms	10/second
3	INT3[b]	DMA Done	DMA Transfer Complete	< 1ms	Varies on transfer length
4	INT2	FEP Interrupt	One or more FEPs requesting service	< 10ms	peak 60/second avg. 10/second
5	INT4	RadMon	Radiation Monitor Asserted	< 10ms	peak 4/second

a. Mongoose Extended Interrupt - Timer EOC (see Section 4.2.5)

b. Mongoose Extended Interrupt - DMA Done (see Section 4.2.5)

4.4 Front End Processor Hardware Interfaces

The following sections describe the Front End Processor hardware interfaces. This information is taken from the “DPA Hardware Specification and System Description, Rev. 04,” by Dorothy Gordon, MIT Drawing Number 36-02104. Additional information is included in advance of future publications of this document, and is marked as TBD.

NOTE: Although the BEP to FEP Command Registers physically reside on each FEP, they are under the control of the BEP and serve as BEP to FEP interface, and are described in Section 4.3.12 .

4.4.1 Memory Map Overview

This section illustrates the overall Front End Processor hardware memory map:

TABLE 8. Front End Processor Memory Map Overview

Virtual Address	Byte Size	Name	Use
0xbfc00000 aliased to portion of Auxiliary Image Data area	0x8000	Boot RAM	Contains Bootstrap loader software (see Section Section 40.0) and Microboot Control Word (see Section 4.2.1). This area MUST be loaded by the Back End Processor via the shared memory interface prior to releasing the FEPs reset line.
0xbf800000	N/A	External Debug Board	Used for developement and pre-flight testing
0xbf400000	0x30	Mongoose CSI Sub-block	see Section 4.2
0xa0c00000	12Mbits	Bias Map Memory	Stores one 12-bit bias value for each pixel in a CCD image. The bias values are set by software.
0xa0800000	12Mbits	Image Memory	Contains 12-bit image pixel values read by the FEPs image acquisition hardware
0xa0401000	0x40	Image Control Registers	see below
0xa0400000	0x18	FEP I/O Devices	see below
0xa0000000	1.5MB	Auxiliary Image Data (Bulk Memory)	Contains scratch data space, overclock pixel buffers, threshold bit-plane, bias parity error plane, BEP-to-FEP command mailbox, FEP-to-BEP request mailbox, FEP-to-BEP ring buffer. A portion of this RAM is also used for the Boot RAM area. If the Boot RAM area is used for another purpose when the system is running, the BEP is required to re-load the area prior to re-booting the FEP.

TABLE 8. Front End Processor Memory Map Overview

Virtual Address	Byte Size	Name	Use
0x80080000	0x20000	Instruction Cache (SEU-immune)	Contains copy of code loaded from the BEP using a program loaded by the BEP into the FEP's Boot RAM.
0x80000000	0x20000	Data Cache (SEU-immune)	Contains data space for run-time stacks, and FEP program data.

4.4.2 Control Register (0xA0400000)

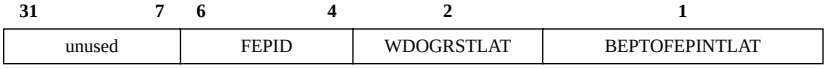
This section describes the Control Register residing on each Front End Processor. This register controls the state of various Front End Processor devices. The format of this register is as follows:

TPLANWENB ..... Enable writes to the threshold plane.

OCLOCKWENB..... Enable writes into the overclock buffer.

4.4.3 Status Register (0xA0400004)

This section describes the Status Register residing on each Front End Processor. This register supplies the current state of various Front End Processor devices. The format of this register is as follows:



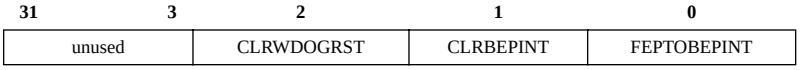
BEPTOFEPINTLAT ..... Indicates interrupt received from BEP.

WDOGRSTLAT..... Indicates if last reset was caused by watchdog timeout.

FEPID..... FEP id.

4.4.4 Pulse Register (0xA0400008)

This section describes the Pulse Register residing on each Front End Processor. This register resets various status and interrupt conditions. The format of this register is as follows:



FEPTOBEPINT..... Generates interrupt to BEP.

CLRBEPINT ..... Clear BEP interrupt latch.

CLRWDOGRST ..... Clears WDOGRSTLAT in FEP status register.

4.4.5 FEP Image Buffer (0xA0800000

Each Front End Processor contains a region of memory into which the image acquisition hardware stores incoming pixels. The buffer can hold a 1024 x 1024 pixel image. Each pixel is stored as a 12-bit value. From the software point of view, 16-bit reads from a given

location return the pixel pulse height in the least significant 12-bits, and zeros in the remaining 4-bits. 32-bit reads return two pixels. The least significant half of the word represents one pixel in the row, and the most-significant half represents the pixel in the next column (or the first pixel in the next row if at the end of a row).

Whether or not an image is written to the image buffer by the hardware is controlled using the “enable next image” flag in the Image Pulse Register (see Section 4.4.15). If the “enable next image” flag is asserted when a new image arrives at the FEP, the hardware de-asserts the flag, and starts writing the new image into the image buffer. If the flag is de-asserted when a new image arrives, the hardware silently discards the image data until the next image. The hardware continues to discard data until an image arrives and the flag has been asserted by the software.

4.4.6 FEP Pixel Bias Map (0xA0C00000)

Each Front End Processor contains a region of memory which is used to contain 12-bit pixel bias values. Each pixel bias value in the map corresponds to a pixel in the image buffer. Prior to processing data, the software running on the FEP computes the bias level for a given pixel and stores the bias value into this map. As the hardware acquires pixel data, it subtracts the bias value from the pixel data and compares the result to the value contained in a threshold register. If the pixel minus its bias value is greater than the threshold register value, the hardware sets a bit in a threshold plane corresponding to the pixel.

Each 16-bit location within the bias map corresponds to 1 pixel. The least-significant 12-bits are the bias value for the corresponding pixel. The next significant bit is a copy of the parity bit for the bias map value. The most significant bit in the 16-bit word is the parity error bit. If this bit is set, then one or more of the bits within the pixel bias map entry, or the pixel parity bit, have been corrupted (see Section 4.4.7).

4.4.7 FEP Pixel Bias Parity Plane

Since pixel bias map memory is vulnerable to single-event upsets (SEUs), and that pixel bias values are used over an entire science run, the hardware provides a pixel bias parity plane. Each bit corresponds to one pixel bias map location. The memory for the plane can be located anywhere within Auxiliary Image Data memory on a 64K byte boundary, and is set by the software using the Bulk Memory Segment Allocation Register Section 4.4.9 . The hardware uses ODD parity, meaning that if there are an even number of ‘1s’ in a given pixel bias map value, the corresponding parity bit is set to ‘1’ to make the total come out to an odd number. If there are an odd number of ‘1s’, then the parity bit is ‘0.’

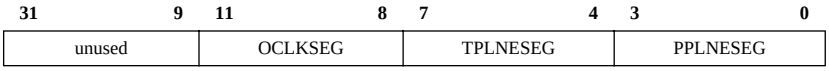
4.4.8 FEP Threshold Plane

Each pixel in the image buffer has a corresponding bit in the threshold bit plane. The location of the threshold plane can be anywhere within the Auxiliary Image Data memory, and is set by software via the Bulk Memory Segment Allocation Register Section 4.4.9 . As images are acquired by the hardware, the threshold logic compares the pixel value minus

its bias map value, to the threshold register associated with the pixel's column. If the adjust pixel pulse height exceeds the threshold register level, the hardware sets the corresponding bit in the threshold plane to '1.' Otherwise, it sets the bit to '0.' The software uses the threshold plane to quickly search for candidate events, 32 at a time.

4.4.9 Bulk Memory Segment Allocation Register (0xA0400008)

This section describes the Bulk Memory Segment Allocation Register residing on each Front End Processor. This register designates which segment of the bulk memory contains the bias map parity plane, the pixel threshold plane and the overclock buffer. The format of this register is as follows:



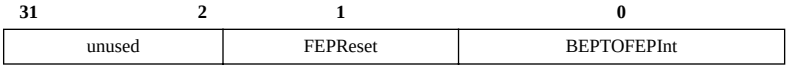
PPLNESEG ..... Page where parity plane is stored.

TPLNESEG ..... Page where threshold plane is stored.

OCLKSEG ..... Page where overclock buffer is stored.

4.4.10 BEPtoFEP Command Register (0xA040000C)

This section describes the BEPtoFEP command register. This register is actually controlled by the BEP and is read-only to the Front End Processor. It allows the BEP to reset and interrupt a particulare FEP. The format of this register is as follows:



BEPTOFEPInt ..... Generate an interrupt in the FEP.

FEPReset ..... Reset the FEP.

4.4.11 Latched Timestamp Register (0xA0400020)

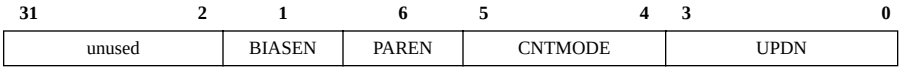
This register contains a 24-bit latched timestamp of the FEP slave counter. It is set upon the detection of a beginning of frame from the DEA sequencer.

4.4.12 Running Timestamp (0xA04000014)

This register contains a 24-bit running timestamp of the FEP slave counter.

4.4.13 Image Control Register (0xA0401000)

This section describes the Image Control Register residing on each Front End Processor. This register controls the state of various Front End Processor devices relating to image processing. The format of this register is as follows:



UPDN ..... Count direction of four video chain (VC) cameras. 1 bit per counter, 0 indicates "count up", 1 indicates "count down".

CNTMODE ..... Number of VC counters used to calculate the image address. (00 - use 1, 01 use 2, 1x - use 4)

PAREN ..... Enabel parity checking

BIASEN ..... Enable use of bias map.

4.4.14 Image Status Register (0xA0401004)

This section describes the Image Status Register residing on each Front End Processor. This register supplies the current state of various Front End Processor devices related to the image processing. The format of this register is as follows:

31 5	4	3	2	1
unused	NXTACQENB	SQERROR	PARERRPR	BOFINT

BOFINT ..... Beginning of Frame interrupt.

PARERROR ..... Parity error detected.

SQERROR ..... Sequencer error detected.

NXTACQENB..... Feedback of corresponding pulse register bit.

4.4.15 Image Pulse Register (0xA0401004)

This section describes the Pulse Register residing on each Front End Processor. This register resets various status and interrupt conditions. The format of this register is as follows:

31 4	3	2	1	0
unused	ARMNXTACQ	CLRSEQERROR	CLRPARERROR	CLRBOFINT

CLRBOFINT..... Clears BOF interrupt latch..

CLRPARERROR ..... Clears Parity Error latch.

CLRSEQERROR ..... Clears Sequencer Error latch.

ARMNXTACQ..... Enable hardware to capture next image.

4.4.16 FEP Threshold Registers (0xA0401010)

Each CCD has four output nodes, each of which feed an associated video chain within the DEA. Within an acquired image, some columns of pixels are produced from one output node, the next group of columns another, and so on. Each Front End Processor has four threshold registers, one for each video chain. As images are acquired by the FEP hardware, their adjusted pixel values are compared with the contents of the threshold register corresponding to the node which produced the pixel. The threshold value is a 14-bit 2’s complement number.

4.4.17 StartCnt Registers (0xA0401020)

There are four start counts, one per video chain (VC) counter. The start count is a 10-bit unsigned value.

4.4.18 Image Start Address (0xA0401030)

The image start address is a 10-bit value which determines where the address generator starts writing values into the image memory for each exposure.

4.4.19 CCD Start Address (0xA0401034)

The CCD start address is a 10-bit value which determines how many rows, if any, of a CCD image frame that should be skipped before enabling active image acquisition.

4.4.20 CCD Num Rows (0xA0401038)

The CCD number of rows is a 10-bit value which determines how many rows of a CCD image frame are included in the active image frame.

4.4.21 CCD Select (0xA040103C)

The CCD select is a 4-bit value which indicates which CCDs are active. A value of 0-9 selects one of the 10 CCD inputs driven by the DEA. A value of A-F disables image acquistion.

4.4.22 Interrupts

This section describes the Front End Processor Interrupts. The Front End Processor hardware supports three external interrupts, and all Mongoose Extended Interrupts. The software running on a Front End Processor expects to use only two of these interrupts. The following table lists the interrupts used by the Front End Processors, interrupt input assignments and timing requirements of each interrupt:

TABLE 9. Front End Processor Interrupts

Input	Name	Causes	Response Time	Frequency
INT0	BEPINT	Back End Processor (not currently used)	-	-
INT1	BOFINT	DEA PRAM-generated Beginning of Image Frame	< 0.1ms	peak 10/second
INT2	SEQERR	Sequencer Error Detected (not currently used)	-	-
INT3[a]	DMA Done	DMA transfer complete	< 0.1ms	Varies on tranfser length
INT4	PARERR	Bias Pixel Map Parity Error (not currently used)	-	-

a. Mongoose Extended Interrupt - DMA Done (see Section 4.2.5)

4.5 Nucleus Real-Time Executive Functions and Formats

This section lists the interfaces to Nucleus RTX, by Accelerated Technologies Inc. The information presented in this section is derived from the “Nucleus RTX Reference Manual,” by Accelerated Technology, Inc.

The structure definitions and function prototypes are supplied with the Nucleus source code distribution, under a source code license agreement with MIT, and are not provided in this document.

4.5.1 Configuration and Startup

This section describes how Nucleus RTX is configured during startup. Nucleus RTX is configured using a set of named data structures, initialized prior to starting Nucleus. In the RTX documentation, the provided examples initialize these structures are initialized at compile time, although there is no reason they cannot be filled at run-time, prior to starting the executive.

The first data location that Nucleus RTX may use to allocate its structures during startup is delimited by the address of the unsigned pointer variable *`IN_Last_Address_Used`*. After startup, the contents of *`IN_Last_Address_Used`* indicates how much memory has been used by Nucleus. The unsigned pointer variable *`IN_Last_Memory_Address`* points to the very last memory location available to Nucleus RTX.

NOTE: If the dynamic heap feature of Nucleus is used, after initialization, Nucleus uses the space between the locations pointed to by the *`IN_Last_Address_Used`* and *`IN_Last_Memory_Address`* as its general purpose memory heap. ACIS does not use this feature.

4.5.1.1 Task Setup

Nucleus RTX tasks are configured using an array of Task Definition Structures named *`IN_System_Tasks`*. Each structure entry within the array defines a Nucleus RTX task, where the array index of the entry defines the RTX Task Id of the corresponding task. Each entry defines the starting address of the task, the size of the task’s stack in 32-bit words, the priority of the task, whether or not to start the task upon startup, whether or not the task allows preemption when it is started, and whether or not to time-slice the task, and if so, how many timer-ticks to allow the task to run in its slice. The end of the array is delimited by an entry whose task address is 0.

4.5.1.2 Memory Pool Setup

Nucleus RTX fixed size memory pools are configured using an array of Fixed Partition Structures named *`IN_Fixed_Partitions`*. Each structure entry within the array defines a Nucleus RTX fixed size pool of memory buffers, where the array index of the entry defines the RTX Pool Id of the corresponding pool. Each entry specifies the number

of 32-bit words in a single buffer in the pool, and the number of buffers to maintain in the pool. The end of the array is delimited by a entry which specifies zero for the buffer size.

4.5.1.3 Queue Setup

Nucleus RTX queues support a pre-configured number of fixed size items. The queues are configured using an array of Queue Definition Structures named *`IN_System_Queues`*. Each structure entry within the array defines a Nucleus RTX queue, where the array index of the entry defines the RTX Queue Id of the corresponding queue. Each entry specifies the number of items which can be held by the queue, and the size of an item, in 32-bit words. The end of the array is delimited by a entry which specifies a queue capacity of zero items.

4.5.1.4 Event Group Setup

Nucleus RTX event groups are configured by specifying the number of desired groups in the *`IN_System_Event_Groups`* unsigned integer variable. The event group identifiers will range from 0 to (*`IN_System_Event_Groups`* - 1).

4.5.1.5 Semaphore Setup

Nucleus RTX semaphores are configured by specifying the number of desired semaphores in the *`IN_System_Resources`* unsigned integer variable. The semaphore identifiers will range from 0 to (*`IN_System_Resources`* - 1).

4.5.2 Task Management Functions

This section lists and briefly describes each of the Nucleus functions used to manage tasks.

4.5.2.1 NU_Change_Priority()

Inputs. Nucleus Task Id
New Priority

Outputs. Success or error code

Description.

This function changes the priority of the task specified by the Task Id.

4.5.2.2 NU_Change_Time_Slice()

Inputs. Nucleus Task Id
New Time Slice

Outputs. Success or error code

Description.

This function changes the time-slice of the task specified by the Task Id.

4.5.2.3 NU_Control_Interrupts()

Inputs. Whether to enable or disable interrupts

Outputs. Previous interrupt state

Description.

This function enables or disables interrupts, returning the previous interrupt state.

4.5.2.4 NU_Current_Task_Id()

Inputs. None

Outputs. Task Id of currently running task

Description.

This function returns the id of the currently running task, or the task which was interrupted if this function is called within an interrupt handler.

4.5.2.5 NU_Disable_Preemption()

Inputs. None

Outputs. None

Description.

This function disables preemption of the currently running task by another task. Calls to this function may made more than once. To re-enable preemption, an equal number of calls to `NU_Enable_Preemption`() must be made

4.5.2.6 NU_Enable_Preemption()

Inputs. None

Outputs. None

Description.

Each call to this function unwinds one previous call to `NU_Disable_Preemption`(). Once unwound, other tasks may preempt the currently running task.

4.5.2.7 NU_Relinquish()

Inputs. None

Outputs. None

Description.

This function causes the currently running task to yield control to other tasks of the same priority.

4.5.2.8 NU_Reset()

Inputs. Task Id

Outputs. Success or error

Description.

This function resets the state of the task specified by Task Id. The task must be in a stopped state at the time of this call.

4.5.2.9 NU_Retrieve_Task_Status()

Inputs. Task Id

Outputs. Success or error
Running state or reason for suspension
Number of times task has been scheduled

Description.

This function retrieves the current status of the task specified by Task Id.

4.5.2.10 NU_Sleep()

Inputs. Number of Timer Ticks

Outputs. None

Description.

This function causes the current task to suspend for the specified number of Timer Ticks (in ACIS, each timer tick is 1/10 second).

4.5.2.11 NU_Start()

Inputs. Task Id

Outputs. Success or error

Description.

This function causes the task specified by Task Id to resume execution from a stopped state. If the task was never started, or if has been reset via `NU_Reset`() it will start execution from the start address specified in the initial configuration (see Section 4.5.1.1). If it was stopped via a call to `NU_Stop`(), but not reset, it will resume execution from the point at which it was stopped.

4.5.2.12 NU_Stop()

Inputs. Task Id

Outputs. Success or error

Description.

This function causes the active task specified by Task Id to stop execution, until started via `NU_Start`().

4.5.3 Event Management Functions

Nucleus manages events using Event Groups. An individual Event Group can contain 32 distinct event flags (i.e. the number of bits in an unsigned integer).

4.5.3.1 NU_Set_Events()

Inputs. Event Group Id
Operation (AND or OR)
List of Events

Outputs. Success or Error

Description.

This function logically modifies the events in the indicated event group based on the input operation. If an AND operation is specified, the event list is ANDed with the event flags in the group (i.e. events which are not in the list are de-asserted within the group). If an OR operation is selected, the event list is ORed with the event flags in the group (i.e. events in the list are asserted within the group).

4.5.3.2 NU_Wait_For_Events()

Inputs. Event Group Id
Operation (wait for all, wait for one or more)
De-assert events option
List of Events
Time-out Selection (immediate, never, or # ticks)

Outputs. Success, time-out, or error
Received events

Description.

This function suspends the current task until the desired event conditions are met, or until the time-out expires. The input operations include: Wait until all listed events are asserted in the group; wait until one or more of the listed events are asserted in the group. The caller also has the option to de-assert the indicated event(s) once they’ve been detected.

4.5.4 Resource Management Functions

Nucleus refers to semaphores as “resources.”

4.5.4.1 NU_Release_Resource()

Inputs. Resource Id

Outputs. Success or error

Description.

This function releases the indicated resource (semaphore). The resource must have been previously allocated via `NU_Request_Resource`().

4.5.4.2 NU_Request_Resource()

Inputs. Resource Id
Time-out Selection (immediate, never, or # ticks)

Outputs. Success, time-out, or error

Description.

This function suspends the current task until the indicated resource (semaphore) is available, and then allocates the resource.

4.5.4.3 NU_Retrieve_Resource_Status()

Inputs. Resource Id

Outputs. Success or error
Number of tasks requesting resource

Description.

This function returns the number of tasks attempting to obtain the indicated resource. 0 indicates that the resource is available, 1 indicates that the resource is allocated, but no other tasks are waiting for it, and 2 or greater indicates that one or more tasks are suspended, waiting for the resource.

4.5.5 Queue Management Functions

Nucleus manages fixed length first-in/first-out lists of fixed size items using pre-configured queues.

4.5.5.1 NU_Force_Item_In_Front()

Inputs. Queue Id
Item

Outputs. Success or error

Description.

This function copies the supplied item into the front of the identified queue. If the queue is full, the last item in the queue is discarded.

4.5.5.2 NU_Retrieve_Item()

Inputs. Queue Id
Time-out selection (immediate, never, # ticks)

Outputs. Success or error
Copy of de-queued item

Description.

This function suspends the current task until the identified queue has an item. It then copies the first item in the queue and removes it from the queue.

4.5.5.3 NU_Retrieve_Item_Multi()

Inputs. Three Queue Ids
Time-out selection (immediate, never, # ticks)

Outputs. Success, queue(s) empty, time-out, or error
Copy of de-queued item

Description.

This function suspends the current task until one of the three identified queues is not empty, and then attempts to retrieve one item from the queue. The function copies the first item and removes it from the queue.

4.5.5.4 NU_Retrieve_Queue_Status()

Inputs. Queue Id

Outputs. Success or error
Number of items current in queue
Size of a single item in the queue

Description.

This function returns the number of items currently in the identified queue, and the item size supported by the queue (in 32-bit words).

4.5.5.5 NU_Send_Item()

Inputs. Queue Id
Item
Time-out selection (immediate, never, # ticks)

Outputs. Success, queue full, time-out, or error

Description.

This function suspends the current task until the identified queue has room to store an additional item, and then copies the supplied item to the end of the queue.

4.5.6 Memory Partition Functions

Nucleus supplies two types of memory management services. The first is a general-purpose memory heap, which supports allocation of arbitrarily sized buffers from a common area. ACIS does not use this service, and its functions are not described in this document. The second type of management supports a pre-configured number of memory pools, each of which manages a fixed number, of fixed size buffers. The following describe the functions which manipulate these pools.

4.5.6.1 NU_Alloc_Partition()

Inputs. Partition Id
Time-out selection (immediate, never, # ticks)

Outputs. Success, partition empty, time-out, or error
Allocated buffer address

Description.

This function suspends the current task until the identified partition has a buffer available, and then allocates the buffer and returns the buffer address to the caller.

4.5.6.2 NU_Available_Partitions()

Inputs. Partition Id

Outputs. Success or error
Number of available buffers

Description.

This function returns the number of free buffers currently available in the indicated partition.

4.5.6.3 NU_Dealloc_Partition()

Inputs. Allocated buffer address

Outputs. Success or error

Description.

This function releases an allocated buffer back to its partition.

4.5.7 Time Management Functions

Nucleus keeps track of a system timer tick counter, and provides functions to set and read this counter.

4.5.7.1 NU_Read_Time()

Inputs. None

Outputs. Current value of system clock

Description.

This function returns the value of its timer tick counter, the system clock. Within ACIS, this counter increments with each timer-tick interrupt (10 per second).

4.5.7.2 NU_Set_Time()

Inputs. Initial clock value

Outputs. None

Description.

This function stores the indicated value into the system clock. This value is subsequently advanced with each timer-tick interrupt (10 per second). The current value of the clock is obtained using `NU_Read_Time`().

4.5.8 Low-level Functions

In order to provide its services, Nucleus requires that certain functions be invoked at certain times.

4.5.8.1 INP_Initialize()

Inputs. Initialized configuration tables

Outputs. None

Description.

This function starts Nucleus. This function must be invoked prior to using any of the Nucleus service routines, and requires that the Nucleus configuration tables are initialized. This function starts the multi-tasking executive and never returns.

4.5.8.2 SKD_Interrupt_Context_Save()

Inputs. None

Outputs. None

Description.

This function must be called by the ACIS interrupt handler, prior to calling any “C/C++” functions or Nucleus service functions. This function saves the registers associated with the interrupted task. This function is an integral part of the context switching features of Nucleus.

4.5.8.3 SKD_Interrupt_Context_Restore()

Inputs. None

Outputs. None

Description.

This function must be called by the ACIS interrupt handler to restore the previously saved register context, and to return control to the next task scheduled to run. This function restores the registers associated with the task being resumed.

4.5.8.4 CLD_Timer_Interrupt()

Inputs. None

Outputs. None

Description.

This function must be called by the ACIS timer-tick interrupt handler to advance Nucleus's system clock, and to cause Nucleus to perform any time-based scheduling operations.

4.6 Command Packet Formats

The command packet formats for the ACIS instrument software are defined in the “ACIS Instrument Program and Command Language List,” MIT 36-01410.

4.7 Telemetry Packet Formats

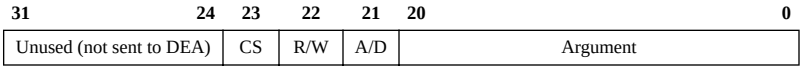
The telemetry packet formats for the ACIS instrument software are defined in the “ACIS Instrument Program and Command List,” MIT 36-01410.

4.8 DEA Command and Status Formats

This section describes the interface to the Detector Electronics Assembly. The information described in this section is taken from the “DPA/DEA Interface Control Document,” MIT 36-02205.

4.8.1 Command Format

DEA command words are 24-bits wide. Each command word takes about 25us to transmit to the DEA. When written into the DEA command port register, the upper 8-bits of the 32-bit word are ignored. The following illustrates the format of commands to the DEA:



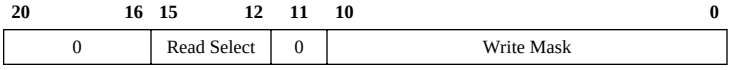
Argument.....This field may contains data, address, or card selection arguments depending on bits 21:23.

A/D .....Address/Data selector [0:Argument is Data, 1:Argument is Address]

R/W .....Read/Write selector [0: Write, 1: Read]

CS .....Card Selection Command [0: Register Command, 1: Card Select Command]

The following illustrates the argument format for Card Selection commands (CS=1):



Write Mask.....This is a bit-field where each bit corresponds to one CCD Controller board. Bit 0 corresponds to board 0, bit 1 to board 1 and so on. When a bit is ‘1’, the corresponding board is enabled, and will execute subsequent register sent to the DEA. When a bits is ‘0’ the corresponding board ignores subsequent register commands sent to the DEA.

Read Select.....This field selects which DEA board (CCD Controller, or System Control Board) will respond to a subsequent query. Only one such board can be enabled at a time.

The following table maps combinations of CS, R/W and A/D to specific command types:

TABLE 10. Command Types

CS	R/W	A/D	Argument	Command Type
0	0	0	Data	Write “argument” value to device/memory location indexed by address register
0	0	1	Address	Write “argument” to address register
0	1	0	Unused	Read data from device/memory location indexed by address register
0	1	1	Unused	Read back address from address register
1	0	X	Card Select	Write card select word into all powered DEA boards
1	1	X	Card Select	Read back card select from DEA board indexed by Read Select field.

4.8.2 Status Format

Status words, sent by one of the DEA boards in response to a command, are 24-bits in length. The response time to a command requesting status can be up to 100us. When read from the 32-bit DEA Status Port register, the unused upper 8-bits will have an unspecified value. The detailed format of a particular status response depends on the device being queried on the DEA.

4.8.3 DEA Boards and Devices

4.8.3.1 Board Register Memory Map

Each DEA board has a collection of memory-mapped registers and RAM. Each of these registers or locations is selected by writing to the board's address register (see Section 4.8.1). The addressed register or location is then read or written using a subsequent command. The following illustrates the DEA board memory map:

TABLE 11. DEA Memory Map

Device/ Memory Section	Address Range	Description
SRAM	0:0x7fff	Sequencer RAM (see Section 4.8.3.5 for the word format)
PRAM	0x8000:0xffff	Program RAM (see Section 4.8.3.4
Subsystem Registers	0x10000:0x1003f	Collection of various control/status registers
DAC Registers	0x10040:0x1005f	Collection of Digital-to-Analog Converters (Write-only)
Housekeeping Registers	0x10008:0x100ff	Collection of Housekeeping devices (Read-only)

4.8.3.2 Digital-to-Analog Converters

The command and status formats for the DACs are TBD.

4.8.3.3 Housekeeping Query Command and Status Formats

The Housekeeping command and status formats are TBD.

4.8.3.4 PRAM Format

The Sequencer Program RAM (PRAM) consists of an array of pairs of 16-bit words, organized into blocks. Each block starts with a header block, which specifies the number of times to repeat the block, and the number of subsequent pairs of words belong to the block. The format of this header is as follows:

15	14	13	12	11 ... 0
1	1	Option		PRAM Block Repeat Count
1	0	Page Jump		Couplet Count

PRAM Block Repeat Count.....This specifies the number of times to repeat the entire block

Option.....This specifies the next sequence option [0: Restart, 1:Continue, 2:Halt, 3:Page Jump]

Couplet Count .....This specifies the number of PRAM word pairs (couplets) following the block

Page Jump .....If Option is 3, this specifies the PRAM page to jump to

The following illustrates the format of the PRAM couplets within a block:

15	14	13 ... 5	4	3 ... 0
0	1	SRAM Page Address	0	PixCode
0	0	0 0		Major Cycle Count

PixCode.....This code is sent to the Front End Processor with each major cycle (pixel). Its values are TBD.

SRAM Page Address.....This specifies which block of 64 SRAM blocks should be sequenced during a major cycle.

Major Cycle Count.....This specifies how many times to repeat the selected SRAM block

4.8.3.5 SRAM Format

SRAM words are organized int blocks of 64 16-bit words. Each bit of the 16-bit word drives the state of one of the CCD or DEA clock signals. Spread over the period of 1 major cycle (1 pixel), the sequencer steps through each word in the 64 word block, driving the states of the clocking signals.

The bit assignments for the SRAM words are TBD.

4.9 BEP to FEP Communication Protocol

The following sections describe the communication protocol used between the Back End and Front End Processors.

4.9.1 Shared Memory Interface

This section describes the shared memory interface between the Back End and Front End Processors. Sections of each Front End Processor's memory, and some of its device ports, are mapped into the Back End Processor's address space. These memory sections are used by the Back End Processor to load and run code in the FEPs, and to interactively communicate with the FEPs while processing science.

The FEP sections and addresses, and the corresponding BEP addresses are TBD.

4.9.2 FEP Boot Procedure

This section describes the procedure used by the Back End Processor to start up each of the Front End Processors.

To start a Front End Processor, the Back End Processor performs the following steps:

1. Power on the FEP (if it does not already have power). The FEP will come up in a reset state. Wait for at least 100ms before performing any other actions.
2. If the power was already on, assert the reset line to the FEP (if is not already asserted)
3. Load the Microboot Control Word to the last shared memory location of the FEP's Boot RAM
4. Load boot start-up code to the reset vector location in the FEP's Boot RAM (0xbfc00000)
5. Load the remainder of the boot-strap code into shared memory.
6. Release the FEPs reset line to run the loaded boot-strap code.
7. Complete the load using the running boot code.

NOTE: If the FEP's Watchdog timer expires, the FEP will assert and hold its reset line and generate an interrupt on the BEP. The BEP can then determine what action, if any, it needs to take to restart the FEP.

4.9.3 BEP to FEP Command Mailbox

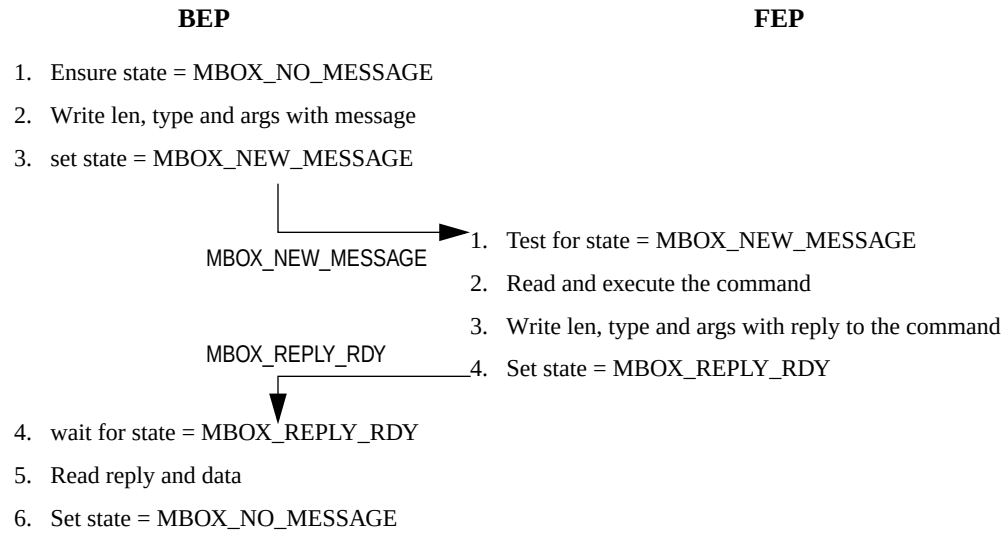
This section describes the format of the BEP to FEP Command Mailbox. This mailbox is used to pass commands from the BEP to the FEP, and the FEP's reply to the command back to the BEP. The software on each Front End Processor establishes one of this type of mailbox at address TBD. The total length of the mailbox is 512 bytes. The format of this mailbox is as follows:

TABLE 12. BEP to FEP Command Mailbox Format

Field	Type	Range	Description
state	enum	MBOX_NO_MESSAGE = 0, MBOX_NEW_MESSAGE = 1, MBOX_REPLY_RDY = 2	This field indicates the current state of the mailbox.
len	unsigned	1:129	This field contains the length of the remainder of the message, in units of 32-bit words.
type	int	TBD	This field identifiers the type of message and distinguishes whether the command is handled by the FEP's I/O library, or by the FEP's Science Processing code.
args	unsigned[128]	-	This array contains any arguments associated with "type."

NOTE: A detailed description of how science uses this mailbox is provided in Section 4.10 .

The dialog scenario for using the Command Mailbox is as follows:



4.9.4 BEP to FEP Command and Reply Formats

Detailed descriptions of FEP I/O library command and reply formats are provided in Section 39.0 . Detailed descriptions of the FEP Science command and reply formats are provided in Section 4.10 .

4.9.5 FEP to BEP Science Data Ring-Buffer

This section describes the structure of the FEP to BEP Science Data Ring-buffer. This buffer contains an array of TBD record structures, each of which consists of an array of 32 words.

TABLE 13. FEP to BEP Science Ring-Buffer Format

Field	Type	Range	Description
Write Index	unsigned	0:records in ring buffer - 1	This field is an index into the ring buffer data area just beyond the last written location. This index wraps to 0 when it reaches the end of the buffer. This field must only be modified by the FEP. If this value is equal to the Read Index, then the ring-buffer is empty. If this value is one less than the Read Index (or at its greatest value if the Read Index is 0), then the ring buffer is full, and no more data can be written.
Read Index	unsigned	0:records in ring buffer - 1	This field is an index into the ring buffer data area just beyond the last read location. This index wraps to 0 when it reaches the end of the buffer. This field must only be modified by the BEP.
Data Buffer	record[TBD] : where each record is unsigned[32]	-	This is an array of records, where each record is 32 words in length (32-bit words). This buffer must only be modified by the FEP.

4.10 FEP to BEP Science Protocols and Formats (36-53204.03 B)

4.10.1 Purpose

This chapter describes the Science-level interface between the FEPs and the BEP, as defined in the *fepBep.h* include file. The details of the interface itself are further described in Section 39.0 on page 1171. Here we concentrate on those aspects that directly concern the science processes executing within the FEP.

4.10.2 Uses

The ***fepBep*** interface is used in the following situations:

Use 1:: BEP tells a FEP to start a bias calibration or science run.
Use 2:: BEP tells a FEP to stop a bias calibration or science run.
Use 3:: FEP reports an exposure record to the BEP.
Use 4:: FEP reports an event record to the BEP.
Use 5:: FEP reports a raw pixel record to the BEP.
Use 6:: FEP reports a raw pixel histogram to the BEP.
Use 7:: FEP reports the end of exposure record to the BEP.
Use 8:: FEP reports a fiducial pixel record to the BEP.
Use 9:: FEP reports a bias parity error to the BEP.
Use 10:: BEP loads a parameter block into a FEP.
Use 11:: BEP temporarily suspends and resumes processing in a FEP.
Use 12:: BEP asks a FEP to report its current processing status.
Use 13:: BEP loads one or more fiducial pixel addresses into a FEP.

4.10.3 Organization

FEP exposures, events, fiducial pixels, and bias errors are reported to the BEP via the FEP's ring buffers. All other communication uses the BEP-FEP mailbox (see Section 39.5 on page 1175).

4.10.3.1 Ring Buffer Records

All ring buffer records are an integral number of 32-bit words in length, and start on a 32-bit boundary. Their first 32-bit field is a `fepRingType` code indicating their contents,

```
typedef enum {
  FEP_EXPOSURE_REC,       /* output is FEPexpRec */
  FEP_EXPOSURE_END_REC,   /* output is FEPexpEndRec */
  FEP_EVENT_REC_3x3,      /* FEPeventRec3x3 */
  FEP_EVENT_REC_5x5,      /* FEPeventRec5x5 */
  FEP_EVENT_REC_RAW,      /* FEPeventRecRaw */
  FEP_EVENT_REC_HIST,     /* FEPeventRecHist */
```

```
  FEP_EVENT_REC_1x3,     /* FEPeventRec1x3 */
  FEP_FID_PIX_REC,       /* FEPfidPixRec */
  FEP_ERROR_REC          /* output is FEPerrorRec */
} fepRingType;
```

The transfer itself is performed by a call to ***FIOappendBlock()***.

4.10.3.2 Exposure Start Record

When the FEP software detects the arrival of a new image frame from the DEA, it appends a copy of the `FEPexpRec` structure to the ring buffer.

```
typedef struct {
  fepRingType type;          /* = FEP_EXPOSURE_REC */
  unsigned expnum;           /* exposure number */
  unsigned timestamp;        /* time stamp */
  unsigned short bias0[4];   /* initial overclocks */
  short dOclk[4];            /* changes in overclock */
} FEPexpRec;
```

`expnum` — the frame counter value of this new exposure, as reported by the FEP hardware.

`timestamp` — the value of the 1 MHz system clock as latched by the FEP hardware upon arrival of the first pixel of the new frame.

`bias0[4]` — the average DEA output node overclock values that were derived from the initial frame of the most recent bias calibration.

`dOclk[4]` — the 4 overclock correction factors to be applied to each DEA output node of the new frame, where index 0 corresponds to Output Node A, 1 to Output Node B, 2 to Output Node C, and 3 to Output Node D (NOTE: If any of the output nodes are unused for a given mode, the corresponding overclock correction factor is also unused, and is set to 0). Each of these values consists of a raw computed overclock value for the output node, minus the corresponding `bias0`.

4.10.3.3 Exposure End Record

When the FEP ends its processing of an image frame, it appends a copy of the `FEPexpEndRec` structure to the ring buffer.

```
typedef struct {
  fepRingType type;        /* = FEP_EXPOSURE_END_REC */
  unsigned expnum;         /* exposure number */
```

```
  unsigned thresholds;     /* # of threshold crossings*/
  unsigned parityerrs;     /* # of bias parity errors */
} FEPexpEndRec;
```

`expnum` the frame counter value of this new exposure, as reported by the FEP hardware.

`thresholds` the number of threshold crossings detected in the frame.

`parityerrs` the number of bias parity errors detected in the frame.

4.10.3.4 3x3 Event Record

When the FEP software detects a threshold event, it appends a copy of the `FEPeventRec3x3` structure to the ring buffer.

```
typedef struct {
  fepRingType type;         /* = FEP_EVENT_REC_3x3 */
  unsigned short row, col;  /* center pixel address */
  unsigned short p[3][3];   /* pixel values */
  unsigned short b[3][3];   /* bias values */
} FEPeventRec3x3;
```

`row` the row index of the center pixel. The first row in FEP image memory has row index 0, and the index increases by 1 for each subsequent row.

`col` the column index of the center pixel. The first pixel from the first DEA output node has column index 0, and the index increases by one for each subsequent pixel.

`p[3][3]` the 9 pixel values, `p[rows][cols]`, including the central value, `p[1][1]`. Each 12 low-order bits contain the pixel value, with the remaining 4 high-order bits set to zero.

`b[3][3]` the 9 bias values, `b[rows][cols]`, including the central value, `b[1][1]`. Each 12 low-order bits contain the bias value, with the remaining 4 high-order bits set to zero. A bias value of 4094 indicates that the original bias value has suffered a parity error since the most recent bias calibration. A bias value of 4095 indicates that the corresponding image pixel is a member of the ***Bad Pixel List***.

4.10.3.5 5x5 Event Record

When the FEP software detects a threshold event, it appends a copy of the `FEPeventRec5x5` structure to the ring buffer.

```
typedef struct {
  fepRingType type;         /* = FEP_EVENT_REC_5x5 */
  unsigned short row, col;  /* center pixel address */
  unsigned short p[3][3];   /* pixel values */
  unsigned short b[3][3];   /* bias values */
  unsigned short pe[16];    /* edge pixel values */
  unsigned short be[16];    /* edge bias values */
} FEPeventRec5x5;
```

`row` — the row index of the center pixel. The first row in FEP image memory has row index 0, and the index increases by 1 for each subsequent row.

`col` — the column index of the center pixel. The first pixel from the first DEA output node has column index 0, and the index increases by one for each subsequent pixel.

`p[3][3]` — the center 9 pixel values, `p[rows][cols]`, including the central pixel, `p[1][1]`. Each 12 low-order bits contain the pixel value, with the remaining 4 high-order bits set to zero.

`b[3][3]` — the center 9 bias values, `b[rows][cols]`, including the central bias, `b[1][1]`. Each 12 low-order bits contain the bias value, with the remaining 4 high-order bits set to zero. A bias value of 4094 indicates that the original bias value has suffered a parity error since the most recent bias calibration. A bias value of 4095 indicates that the corresponding image pixel is a member of the ***Bad Pixel List***.

`pe[16]` — the 16 outer pixel values—`pe[0]` is 2 rows and 2 columns before the center, `pe[1]` is two rows and 1 column before it, etc. Each 12 low-order bits contain the pixel value, with the remaining 4 high-order bits set to zero.

`be[16]` — the 16 outer bias values—`be[0]` is 2 rows and 2 columns before the center, `be[1]` is two rows and 1 column before it, etc. Each 12 low-order bits contain the bias value, with the remaining 4 high-order bits set to zero. A bias value of 4094 indicates that the original bias value has suffered a parity error since the most recent bias calibration. A bias value of 4095 indicates that the corresponding image pixel is a member of the ***Bad Pixel List***.

4.10.3.6 1x3 Event Record

When the FEP software detects a threshold event, it appends a copy of the `FEPeventRec1x3` structure to the ring buffer.

```
typedef struct {
  fepRingType type;         /* = FEP_EVENT_REC_1x3 */
  unsigned short row, col;  /* center pixel address */
  unsigned short p[3];      /* pixel values */
  unsigned short b[3];      /* bias values */
} FEPeventRec1x3;
```

`row` — the row index of the center pixel. The first row in FEP image memory has row index 0, and the index increases by 1 for each subsequent row.

`col` — the column index of the center pixel. The first pixel from the first DEA output node has column index 0, and the index increases by one for each subsequent pixel.

`p[3]` — the 3 pixel values, `p[rows]`, including the central value, `p[1]`. Each 12 low-order bits contain the pixel value, with the remaining 4 high-order bits set to zero.

`b[3]` — the 3 bias values, `b[rows]`, including the central value, `b[1]`. Each 12 low-order bits contain the bias value, with the remaining 4 high-order bits set to zero. A bias value of 4094 indicates that the original bias value has suffered a parity error since the most recent bias calibration. A bias value of 4095 indicates that the corresponding image pixel is a member of the ***Bad Pixel List***.

4.10.3.7 Raw Mode Pixel Record

As images are acquired by the FEP, the FEP software appends a copy of the `FEPeventRecRaw` structure to the ring buffer.

```
typedef struct {
  fepRingType type;         /* = FEP_EVENT_REC_RAW */
  unsigned short row;       /* pixel row address */
  unsigned short p[1024];   /* pixel values */
  unsigned short oc[MAX_NOCLK*4];
                            /* overclock pixel values */
} FEPeventRecRaw;
```

`row` — the row index of the center pixel. The first row in FEP image

memory has row index 0, and the index increases by 1 for each subsequent row.

`p[1024]` up to 1024 pixel values. Each 12 low-order bits contain the pixel value, with the remaining 4 high-order bits set to zero.

`oc[]` up to MAX_OCLK overclock pixels per DEA output node. Each 12 low-order bits contain the overclock value, with the remaining 4 high-order bits set to zero.

4.10.3.8 Histogram Mode Record

The FEP reads exposures, accumulating histograms of raw pixel values (`FEP_TIMED_PARM_HIST`). Once the specified number of exposures (`nhist` in `fepParmBlock`) have been processed, the FEP software appends a copy of the `FEPeventRecHist` structure to the ring buffer.

```
typedef struct {
  fepRingType type;         /* = FEP_EVENT_REC_HIST */
  unsigned expfirst;        /* first exposure number */
  unsigned explast;         /* last exposure number */
  unsigned short omin[4];   /* minimum node overclock */
  unsigned short omax[4];   /* maximum node overclock */
  unsigned short omean[4];  /* mean node overclock */
  unsigned ovar[4];         /* node overclock variance */
  unsigned hist[4][4096];   /* pixel histogram */
} FEPeventRecHist;
```

`expfirst` the frame counter value of the first exposure in which the histogram was accumulated, as reported by the FEP hardware.

`explast` the frame counter value of the last exposure in which the histogram was accumulated, as reported by the FEP hardware.

`omin[4]` the minimum value of the overclocks from the 4 CCD nodes while the histogram was being accumulated.

`omax[4]` the maximum value of the overclocks from the 4 CCD nodes while the histogram was being accumulated.

`omean[4]` the mean of the average overclock values from the 4 CCD nodes while the histogram was being accumulated. The average is computed for each node of each frame and these are then averaged over `nhist` frames.

`ovar[4]` the mean of the variances of the overclock values from the 4

CCD nodes while the histogram was being accumulated. The variance is computed for each node of each frame and these are then averaged over `nhist` frames.

`hist[4][4096]` the pixel histogram. The first index denotes the CCD nodes, A-D, and the second the number of 12-bit values found in all rows of the CCD during that particular set of exposures.

4.10.3.9 Fiducial Pixel Record

When fiducial pixels have been defined by a prior `BEP_FEP_CMD_FIDPIX` command, the FEP software appends a copy of the `FEPfidPixRec` structure to the ring buffer whenever that particular fiducial pixel is encountered (in timed exposure event-finding mode only).

```
typedef struct {
  fepRingType type;         /* = FEP_FID_PIX_REC */
  unsigned index;           /* fiducial pixel index */
  unsigned val;             /* fiducial pixel pair */
} FEPfidPixRec;
```

`index` the index of this pair of fiducial pixels in the *`args`* array of row and column addresses passed to the FEP in the most recent `BEP_FEP_CMD_FIDPIX` command (see Section 4.10.4.9 on page 148).

`val` the value of the pair of fiducial pixels—the 12 low-order bits (0-11) contain the value of the even-column-number pixel and bits 16-27 contain the value of the pixel at the same row and one higher column number. The remaining bits are set to zero.

4.10.3.10 Error Record

Immediately the FEP software detects a parity error in the bias map, it appends a copy of the `FEPerrorRec` structure to the ring buffer and immediately changes the stored bias value to 4094. NOTE: if *both* of a pair of bias values in neighboring even/odd columns have experienced a parity error, only a single `FEPerrorRec` record will be generated.

```
typedef struct {
  fepRingType type;         /* = FEP_ERROR_REC */
  unsigned short row, col;  /* pixel address */
  unsigned expnum;          /* exposure number */
  unsigned biasval;         /* the erroneous value */
} FEPerrorRec;
```

`row` the row index of the bad bias pixel. The first row in FEP image

memory has row index 0, and the index increases by 1 for each subsequent row.

`col` — the column index of the bad bias pixel. The first pixel from the first DEA output node has column index 0, and the index increases by one for each subsequent pixel.

`expnum` — the frame counter value of the exposure in which the bias parity error was detected, as reported by the FEP hardware.

`biasval` — the value of the 32-bit register containing one or two bad bias values. Bits 0-11 and 16-27 contain the values, bits 12 and 28 contain the corresponding parity bits, and bits 15 and 31 contains the parity error flags.

4.10.4 BEP-FEP Mailbox Messages

The BEP sends commands to the FEPs using a `COMMAND` mailbox (see Section 39.5 on page 1175). However, this is not accompanied by a hardware interrupt, so the FEP must arrange to make frequent calls to ***FIOgetNextCmd()*** to poll the mailbox for new business. This arrives in the form of a COMMAND structure,

```
typedef struct {
  unsigned len;             /* number of args + type */
  int type;                 /* type of request */
  unsigned args[];          /* data */
} COMMAND;
```

Several BEP-to-FEP commands, signified by negative `type` values, are handled automatically within ***FIOgetNextCmd()***. The remainder must be handled by the science-level code. When the command has been processed, the FEP fills a second COMMAND structure and passes it back to the BEP with a call to ***FIOwriteCmdReply()***.

4.10.4.1 Start Bias Calibration

The BEP signals to the FEP that it is to start a bias calibration by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_BIAS
command.args[0] = (unused)
```

This command should be preceded by a `FEP_CMD_LOAD_PARAM` command to load a parameter block. Otherwise, the FEP won't know how to perform the calibration. The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_BIAS
reply.args[0] = status
```

where *status* is FEP_CMD_NOERR if the bias calibration started successfully, or a particular fepCmdRetCode if it didn't. The codes are defined in *fepBep.h*.

4.10.4.2 Start a Timed Exposure

The BEP signals to the FEP that it is to start a timed exposure run by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_TIMED
command.args[0] = (unused)
```

This command should be preceded by a FEP_CMD_LOAD_PARAM command to load a parameter block. Otherwise, the FEP won't know how to start the run. The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_TIMED
reply.args[0] = status
```

where *status* is FEP_CMD_NOERR if the timed-exposure science run started successfully, or a particular fepCmdRetCode if it didn't. The codes are defined in *fepBep.h*.

4.10.4.3 Start a Continuously Clocked Exposure

The BEP signals to the FEP that it is to start a continuously clocked exposure run by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_CCLK
command.args[0] = (unused)
```

This command should be preceded by a FEP_CMD_LOAD_PARAM command to load a continuous clocking parameter block. Otherwise, the FEP won't know how to start the run. The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_CCLK
reply.args[0] = status
```

where *status* is FEP_CMD_NOERR if the continuously clocked science run started successfully, or a particular fepCmdRetCode if it didn't. The codes are defined in *fepBep.h*.

4.10.4.4 Terminate the Current Mode

The BEP signals to the FEP that it is to stop whatever it is doing (please) by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_STOP
command.args[0] = (unused)
```

The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_STOP
reply.args[0] = status
```

where *status* is `FEP_CMD_NOERR` if the FEP stopped its processing, or a particular `fepCmdRetCode` if it didn't, e.g. because it was idling at the time. The codes are defined in *fepBep.h*.

4.10.4.5 Load a Parameter Block

The BEP transfers a parameter block to the FEP by sending it the following COMMAND structure:

```
command.len = 18
command.type = BEP_FEP_CMD_PARAM
command.args[0] = type
command.args[1] = nrows
...
```

The `args` array actually contains a copy of the following `FEPparmBlock` structure.

```
typedef struct {
  fepParmType type;     /* parameter block type */
  unsigned nrows;       /* ending pixel row */
  unsigned ncols;       /* # of pixels/row/node */
  fepQuadCode quadcode; /* Quadrant (node) selection */
  unsigned noclk;       /* # overclocks/row/node */
  unsigned nhist;       /* # exposures/histogram */
  fepBiasType btype;    /* bias algorithm type */
  int thresh[4];        /* user specified thresholds */
  int bparm[5];         /* bias calibration params */
  unsigned nskip;       /* exposure skip factor */
  unsigned initskip;    /* # initial frames to ignore */
} FEPparmBlock;
```

```
typedef enum {
  FEP_NO_PARM,          /* no param block specified */
  FEP_TIMED_PARM_RAW,   /* timed raw mode */
  FEP_TIMED_PARM_HIST,  /* timed raw histogram */
  FEP_TIMED_PARM_3x3,   /* timed 3x3 events */
  FEP_TIMED_PARM_5x5,   /* timed 5x5 events */
  FEP_CCLK_PARM_RAW,    /* continuous raw mode */
  FEP_CCLK_PARM_1x3,    /* continuous 1x3 events */
} fepParmType;

typedef enum {
  FEP_QUAD_ABCD,        /* All four nodes in use */
  FEP_QUAD_AC,          /* Only A and C nodes in use */
  FEP_QUAD_BD           /* Only B and D nodes in use */
} fepQuadCode;

typedef enum {
  FEP_NO_BIAS,          /* none */
  FEP_BIAS_1,           /* algorithm #1 */
  FEP_BIAS_2            /* algorithm #2 */
} fepBiasType;
```

Note that the same structure is used to define parameters in timed mode and in continuous clocking mode. Once the FEP has loaded the block, it replies to the FEP as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_PARAM
reply.args[0] = status
```

where *status* is `FEP_CMD_NOERR` if the parameter block was successfully loaded, or a particular `fepCmdRetCode` if it wasn't. The codes are defined in *fepBep.h*.

4.10.4.6 Suspend FEP Operations

The BEP signals to the FEP that it is to temporarily suspend its operations by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_SUSPEND
command.args[0] = (unused)
```

The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_SUSPEND
reply.args[0] = status
```

where *status* is FEP_CMD_NOERR if the FEP suspended its current task, or a particular fepCmdRetCode if it didn't, e.g. because it was idling at the time. The codes are defined in *fepBep.h*.

4.10.4.7 Resume FEP Operations

The BEP signals to the FEP that it is to resume its operations by sending it the following COMMAND structure:

```
command.len = 2
command.type = BEP_FEP_CMD_RESUME
command.args[0] = mode
```

where mode is the FEP operating mode that is to resume. The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_RESUME
reply.args[0] = status
```

where *status* is FEP_CMD_NOERR if the FEP resumed its suspended task, or a particular fepCmdRetCode if it didn't, e.g. because it hadn't been suspended. The codes are defined in *fepBep.h*.

4.10.4.8 Command a FEP to return its status

The BEP signals to the FEP that it is to return its current processing status by sending it the following COMMAND structure:

```
command.len = 1
command.type = BEP_FEP_CMD_STATUS
command.args[0] = (unused)
```

The reply sent to the FEP is as follows:

```
reply.len = 4
reply.type = BEP_FEP_REPLY_STATUS
reply.args[0] = mode
reply.args[1] = biasflag
reply.args[2] = biasparityaddr
reply.args[3] = bias0[0] and bias[01]
reply.args[4] = bias0[2] and bias0[3]
```

where *mode* is the command.type of the command that is currently executing in the FEP (or zero if it is idle), *biasflag* is TRUE if the FEP contains a valid bias map, or FALSE if it doesn't, and *biasparityaddr* is the address, in the FEP's address space, of the start of its

bias parity plane. *bias0*[4] is an array of unsigned 16-bit average overclock values from the first frame of the bias calculation, packed into two 32-bit ring buffer words.

4.10.4.9 Load one or more fiducial pixel addresses into a FEP

Fiducial pixels are particular CCD pixels whose values are always to be reported by the FEP in timed exposure event modes. The BEP passes one or more fiducial pixel addresses to the FEP by sending it the following COMMAND structure:

```
command.len = varying
command.type = BEP_FEP_CMD_FIDPIX
command.args[0] = row_and_column_1
command.args[1] = row_and_column_2
command.args[2] = row_and_column_3
...
```

where the column addresses[1] occupy bits 0-11 of each element of the args array, and the corresponding row addresses occupy bits 16-27. The number of fiducial pixels is one less than `command.len`. The reply sent to the FEP is as follows:

```
reply.len = 2
reply.type = BEP_FEP_CMD_FIDPIX
reply.args[0] = status
```

where *status* is `FEP_CMD_NOERR` if the fiducial pixels were successfully loaded, or a particular `fepCmdRetCode` if they weren't. The codes are defined in *fepBep.h*.

1. Since fiducial pixels are always reported in contiguous even/odd pairs, an odd column address will be decremented upon receipt.

5.0 Mongoose and Back End Registers (36-53205 B)

5.1 Purpose

The purpose of the **Mongoose** and **BepReg** (Back End Registers) classes is to provide low-level access to the R3000 System Coprocessor (Coprocessor 0) Registers, the Mongoose Command/Status Interface (CSI) Registers, and the ACIS-specific Back End Processor Registers. See Section 4.1 , Section 4.2 , and Section 4.3 for descriptions of the R3000 core processor, the Mongoose extensions to the R3000 core, and the Back End Processor hardware, respectively.

In addition to these classes, this section also describes the **Leds** and **BootMode** utility classes. The **Leds** class is responsible for providing a layer of abstraction to the software-accessible discrete telemetry bits. The **BootMode** class is responsible for providing the cause of the most recent reset of the Back End Processor.

5.2 Uses

The **Mongoose** class provides the following features:

Use 1:: Read the contents of each of the R3000 System Coprocessor Registers
Use 2:: Write the contents of the R3000 System Coprocessor Status Register
Use 3:: Provide memory-mapped access to all of the Mongoose CSI Registers
Use 4:: Set and clear sets of bits in the Mongoose Configuration Register
Use 5:: Set the DMA mode value in the Mongoose Configuration Register
Use 6:: Set and clear sets of bits in the Mongoose Extended Interrupt Mask Register
Use 7:: Clear latched extended interrupts by writing to the Extended Cause Register
Use 8:: Test user addresses against Instruction Cache and Data Cache boundaries.
Use 9:: Copy information to and from the Instruction Cache

The **BepReg** class provides the following additional features:

Use 10:: Set and clear sets of bits in the Back End's Control Register
Use 11:: Write a value to the software discrete telemetry bits (LEDs) in the Back End's Control Register
Use 12:: Read the contents of the Back End's Status Register
Use 13:: Write sets of bits to the Back End's Pulse Register
Use 14:: Provide memory-mapped access all of the Back End Registers

The **Leds** class provides the following feature:

Use 15:: Set the software discrete telemetry (LED) values

The **BootMode** class provides the following feature:

Use 16:: Indicate the cause of the most recent reset of the Back End Processor

5.3 Organization

With the exception of the **IntrGuard** class (see Section 6.0), the **Mongoose** and **BepReg** classes are stand-alone classes, and do not rely on any higher level class definitions. The only relationship is that the **BepReg** and **Mongoose** class use the **IntrGuard** class to temporarily disable interrupts when performing atomic operations, such as a read-modify-write of a shared register. The **IntrGuard** class, then uses the **Mongoose** class to access the R3000 Status Register needed to perform the interrupt disable and enables. The **Leds** uses the **BepReg** class to set the discrete telemetry code and **BootMode** class uses the **BepReg** class to obtain the contents of the hardware Status Register.

5.4 Class Instances and Register Bit-field Definitions

5.4.1 Class Instances

Access to the Mongoose registers is provided by a global **Mongoose** class instance, named *mongoose* and the BEP register class is accessed via a single global instance *bepReg*.

Since the Mongoose CSI registers are packed into adjacent memory locations, and this will not change over the development cycle, the *mongoose* instance provides a pointer to a structure which directly overlays the CSI block. This provides the lowest level device code direct access to the Mongoose's device control registers.

The Back End Registers are currently NOT packed into adjacent memory addresses, and therefore, this approach is not appropriate. Instead, the Back End Registers module currently provides all of its functions as in-line functions to address of the various BEP registers and atomically manage the control, status and pulse registers.

5.4.2 Register Bit-field Definitions

The R3000 register and bit definitions can be found in the "MIPS Programmer's Handbook, Section A.4 Registers." The Mongoose register and bit definitions can be found in the "Mongoose ASIC Microcontroller Programming Guide, Section 9.0 Command/Status Interface (CSI) Registers." The Back End Processor registers are listed and described in the "DPA Hardware Specification and System Description" in the sections pertaining to the Back End Processors Overall Description and subsequent subsections.

Both the **Mongoose** and **BepReg** classes provide enumerations which define the various bits in each register they support. The bit-definition names are qualified by the name of the class defining them. This provides in-code documentation as to where to find the actual definition. For example, if the **Mongoose** class defines the R3000 status register bit, SR_IEC (Status register Interrupt Enable Current), users of this definition (i.e. all classes other than the **Mongoose** class) must refer to the bit by qualifying with the **Mongoose** class definition, "**Mongoose**::SR_IEC." Rather than list the many definitions in this sec-

tion, refer to Section 4.3 for a detailed description of these bits; the constants are defined in mongoose.H and bepreg.H.

5.5 Scenarios

5.5.1 Mongoose Class Scenarios

5.5.1.1 Use 1: Read the R3000 System Coprocessor Registers

The **Mongoose** class provides functions which read the R3000 System Coprocessor registers using the “mfc0” (Move from Coprocessor 0) assembler instruction (NOTE: The **Mongoose** class uses a small set of assembler functions to access these registers. These support functions are provided by filesstartup/asm_startup.S). This provides the client code with the contents of the following registers:

- Status Register (SR) - This register contains the key R3000 interrupt and processor control bits. This register is primarily used by the Interrupt Controller class (see Section 6.0), and by code which needs to perform atomic operations. It is accessed by the `getStatusReg`() member function.
- Cause Register (CAUSE) - This register contains information pertaining to the cause of an interrupt. This register is primarily used by the Interrupt Controller class. It is accessed by the `getCauseReg`() member function.
- Bad Virtual Address Register (BVADDR) - This register contains the address which caused the last address exception. It is accessed by the `getBadVaddrReg`() member function.
- Exception Program Counter Register (EPC) - This register contains the address to return to after handling the current exception or interrupt. This register is currently only used by the assembly language low-level interrupt handling code. It is accessed by the `getEpcReg`() member function.

5.5.1.2 Use 2: Write the R3000 Status Register

The R3000 Status Register controls interrupt enables, and masks individual interrupt lines on the R3000 core. This, in conjunction with the Mongoose’s Extended Interrupt Mask register, is used to mask and unmask individual interrupt causes, as globally enable and disable interrupts. Client functions modify this register using the `setStatusReg`() member function.

In general, most low-level client functions want to protect themselves from interrupts while read-modify-writing a shared register. The following C++ code fragment illustrates a safe way of accomplishing this with the **Mongoose** class:

```
unsigned oldStatusRegister; // Saved contents of Status Register
// ---- Disable interrupts, saving previous contents ----
oldStatusRegister = mongoose->setStatusReg(Mongoose::SR_DISABLE_INTS);
// code to read/modify and write the shared register
// ---- Restore original Status Register contents ----
mongoose->setStatusReg (oldStatusRegister);
```

5.5.1.3 Use 3: Memory-map access to Mongoose Registers

Since the Mongoose CSI registers are memory-mapped into a contiguous block in memory, the **Mongoose** class provides access to these registers in the form of a pointer to a data structure. The registers are accessed by referring to the register by name through the *regs* register pointer, contained within the global **Mongoose** class instance *mongoose*. The following code fragment illustrates how a client might set the contents of the DMA destination address register:

```
mongoose.regs->dmadst = unsigned(dstptr);
```

5.5.1.4 Use 4: Set and clear bits in Configuration Register

The Mongoose Configuration Registers controls several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **Mongoose** class provides functions which atomically set (`setCfgBits`()) and clear (`clrCfgBits`()) bits in the register.

5.5.1.5 Use 5: Set the DMA mode value

This function, `setDmaMode`(), is a special case of "Use 4: Set and clear bits in Configuration Register." Since setting the DMA mode value in the Configuration involves both setting and clearing bits, this ability is rolled into a separate function specifically for dealing with the DMA mode value. Typically, the DMA code uses this function to start and stop DMA activity.

5.5.1.6 Use 6: Set and clear bits in Extended Interrupt Mask

Just as for the Configuration Register, the Mongoose Extended Interrupt Mask Register controls several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **Mongoose** class provides functions which atomically set and clear bits in the register, `setXmaskBits`() and `clrXmaskBits`() respectively.

5.5.1.7 Use 7: Clear latched extended interrupts

Extended interrupts on the Mongoose are latched. In order to clear a latch for a particular interrupt, one must write a 1 into the corresponding bit in the Extended Cause Register. The **Mongoose** class provides a function which provides this service, `clrXcauseBits`().

5.5.1.8 Use 8: Test addresses against cache boundaries

Since the Mongoose Instruction Cache requires special access, and the Mongoose DMA controller only supports certain types of transfers, the **Mongoose** class provides functions

which test client addresses against the Instruction and Data cache address boundaries (`isIcache`(), `isDcache`() `isUncached`()).

5.5.1.9 Use 9: Copy to and from I-cache

In order to read and write information to and from the Mongoose Instruction cache, one must manipulate the Instruction Cache Address and Data registers. In order to keep block operations to and from I-cache reasonably fast, the **`Mongoose`** class provides block copy functions to and from I-cache (`icacheWrite`(), `icacheRead`()).

5.5.2 BepReg Class Scenarios

5.5.2.1 Use 10: Set and clear bits in BEP Control Register

The BEP's Control Register affects several different devices. As such, access to this register's bits must be shared by more than one client device. To make this sharing easier, the **`BepReg`** class provides functions which atomically set and clear bits in the register (`setControl`(), `clrControl`()).

5.5.2.2 Use 11: Write LEDs

This function is a special case of "Use 10: Set and clear bits in BEP Control Register." Since setting the LED value in the Control Registers involves both setting and clearing bits, this ability is rolled into a separate function specifically for dealing with the LED value (**`BepReg`**::`showLeds`()).

5.5.2.3 Use 12: Read BEP Status Register

In order to provide straight-forward access to the BEP's Status Register, the **`BepReg`** class defines a function which reads and returns the current value of the BEP Status Register (`getStatus`()).

5.5.2.4 Use 13: Pulse bits in BEP Pulse Register

In order to provide straight-forward access to the BEP's Pulse Register, the **`BepReg`** class defines a function which writes a caller-supplied value to the BEP Pulse Register (`pulse`()).

5.5.2.5 Use 14: Memory-map access to BEP Registers

Since most of the **`BepReg`** class functions require access to physical hardware locations, the **`BepReg`** class also provides functions which return type-safe pointers to all BEP registers. These include the following:

- Control Register (read/write)- Contains various device control bits (`ctlReg`)
- Status Register (read only)- This contains various device status bits (`statReg`)
- Pulse Register (write only) - This clears various latched device interrupts and controls (`pulseReg`)
- Command FIFO (read only) - This provides 16-bit command words from the RCTU (`cmdFifoReg`)
- DEA Command Register (write only) - This writes commands to the DEA (`deaCmdReg`)
- DEA Status Register (read only) - This reads status information from the DEA (`deaStatReg`)
- Downlink Controller Start Address (read/write) - This specifies the start address for a telemetry packet transfer (`dtcStartReg`).
- Downlink Controller End Address (read/write) - This specifies the ending address of a telemetry packet transfer (`dtcEndReg`).
- Downlink Controller Address Count (read only) - This specifies the current address of the ongoing telemetry packet transfer(dtcAddrCntReg).
- S/C Counter - Latched (read only) - This contains the timestamp of the last command sent to the DEA.
- S/C Counter - Running (read only) - This contains a running value of the S/C counter.

5.5.3 Leds and BootMode Class Scenarios

5.5.3.1 Use 15: Set the software discrete telemetry values

In order to set the software discrete telemetry (LED) bits to a particular pattern, the client passes the desired pattern to *`leds`*.`show`(), which forwards the passed value to *`bepReg`*.`showLeds`() (see Section 5.5.2.2).

(NOTE: This class allows the **`BepReg`** class to remain an internal member to the ***`Devices`*** class category. The goal is to have the **`BepReg`** class be used directly only by the various device classes.).

5.5.3.2 Use 16: Indicate the cause of the most recent reset

To determine if the most recent reset was caused by a power-on command, the client calls *`bootMode`*.`isPowerOn`(). `isPowerOn`() calls *`bepReg`*.`getStatus`() and tests the power-on status bit. If the most recent reset was caused by a power-on command, `isPowerOn`() returns *`BoolTrue`*. If not, it returns *`BoolFalse`*.

To determine if the most recent reset was caused by the watchdog timer, the client calls *`bootMode`*.`isWatchdog`(). `isWatchdog`() calls *`bepReg`*.`getStatus`() and tests the

watchdog-reset status bit. If the most recent reset was caused by a timeout of the watchdog timer, `isWatchdog`() returns *`BoolTrue`*. If not, it returns *`BoolFalse`*.

5.6 Class Mongoose

Documentation:

This class provides the lowest-level interface to the Mongoose Processor.

NOTE: Some member functions are sequential, therefore access to these functions must be coordinated between active threads to avoid contention. This drives the "Guarded" concurrency attribute listed below. Refer to the "Mongoose Programming Guide" and the "MIPS Programmer's Handbook" for detailed descriptions of the hardware registers provided by the Mongoose Microcontroller.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:

```
clrCfgBits()
clrXCauseBits()
clrXMaskBits()
delay()
getBadVaddrReg()
getCauseReg()
getEpcReg()
getStatusReg()
getXCauseReg()
icacheRead()
icacheWrite()
isDcache()
isIcache()
isUncached()
setCfgBits()
setDmaMode()
setStatusReg()
setXMaskBits()
```

Concurrency: Guarded

Persistence: Persistent

5.6.1 clrCfgBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears bits, designated by 1's in the *mask* argument, in the Configuration Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.2 clrXCauseBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *xcause*

Documentation:

This function writes *xcause* to the Mongoose Extended Interrupt Cause Register. This has the effect of clearing any extended interrupts corresponding to 1's in the *xcause* argument. Interrupts corresponding to 0's are not affected.

Concurrency: Synchronous

5.6.3 clrXMaskBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears the bits, indicated by 1's in the *mask* argument, in the Mongoose's Extended Interrupt Mask Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.4 delay()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *useconds*

Documentation:

This function consists of a busy-loop which iterates for at least *useconds* microseconds. The loop executes without disabling interrupts or executive context switching, so the actual delay introduced by calling this function may be longer.

Preconditions:

useconds must be <= (2^{32} -1)/ITERATIONS_PER_USEC where ITERATIONS_PER_USEC is 5 (TBD)

Concurrency: Synchronous

5.6.5 getBadVaddrReg()

Public member of: **Mongoose**

Return Class: **void***

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Bad Virtual Address Register, using `asm_getBadVaddrReg`().

Concurrency: Synchronous

5.6.6 getCauseReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Interrupt Cause Register, using `asm_getCauseReg`().

Concurrency: Synchronous

5.6.7 getEpcReg()

Public member of: **Mongoose**

Return Class: **unsigned***

Documentation:

This function returns the contents of the R3000 Coprocessor 0's Exception Program Counter Register, using `asm_getEpcReg`().

NOTE: The Exception Program Counter is overwritten by nested interrupts, hence the Sequential concurrency qualifier.

Concurrency: Sequential

5.6.8 getStatusReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

Read and return the contents of the R3000 Coprocessor 0's Status Register, using `asm_getStatusReg`().

Concurrency: Synchronous

5.6.9 getXCauseReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Documentation:

This function returns the contents of the Mongoose Extended Cause Register.

Concurrency: Synchronous

5.6.10 icacheRead()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

```
const unsigned* srcaddr
unsigned* dstaddr
unsigned wordcnt
```

Documentation:

Copy *wordcnt* 32-bit words from the Mongoose Instruction cache, located at *srcaddr* into the buffer pointed to by *dstaddr*.

NOTE: This function does not disable interrupts while reading and writing the I-cache address and data registers, hence the “Sequential” concurrency qualifier.

Concurrency: Sequential

5.6.11 icacheWrite()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

```
const unsigned* srcaddr
unsigned* dstaddr
unsigned wordcnt
```

Documentation:

This function writes *wordcnt* words from the data address *srcaddr* to *dstaddr* within Instruction cache.

NOTE: This function does not disable interrupts while reading and writing the I-cache address and data registers, hence the “Sequential” concurrency qualifier.

Concurrency: Sequential

5.6.12 isDcache()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the boundaries of the Data Cache.The function returns *BoolTrue* if the pointer is within D-cache and returns *BoolFalse* if it is not.

5.6.13 isIcache()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the address boundaries of the Instruction Cache. If the pointer is within I-cache, the function returns *BoolTrue*, else it returns *BoolFalse*.

5.6.14 isUncached()

Public member of: **Mongoose**

Return Class: **Boolean**

Arguments:

```
const void* addr
```

Documentation:

This function tests the passed pointer, *addr*, against the start of non-cached memory. It returns *BoolTrue* if the pointer is beyond cache address space, else *BoolFalse*.

5.6.15 setCfgBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets bits, designated by 1's in the *mask* argument, in the Mongoose Configuration Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.6.16 setDmaMode()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mode*

Documentation:

This function writes the mode value, specified by *mode*, into the Mongoose Configuration Register. Only the DMA Mode bits are affected.

Concurrency: Synchronous

5.6.17 setStatusReg()

Public member of: **Mongoose**

Return Class: **unsigned**

Arguments:

unsigned *value*

Documentation:

This function writes *value* to the contents of the R3000 Coprocessor 0's Status Register, and returns the old contents of the register, using `asm_setStatusReg`()

Concurrency: Synchronous

5.6.18 setXMaskBits()

Public member of: **Mongoose**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets the bits, indicated by 1's in the *mask* argument, in the Mongooses Extended Interrupt Mask Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7 Class BepReg

Documentation:

This class provides the lowest-level interface to the Back End Processor registers.

NOTE: Some member functions are sequential, therefore access to these functions must be coordinated between active threads to avoid contention. This drives the “Guarded” concurrency attribute listed below.

Refer to the “DPA Functional Description and Requirements” for detailed descriptions of the hardware registers provided by the Back End Processor.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:

```
clrControl()
cmdFifoReg()
ctlReg()
dtcAddrCntReg()
dtcEndReg()
dtcStartReg()
deaCmdReg()
deaStatReg()
getControl()
getStatus()
pulseReg()
pulse()
scCntLatTimeReg()
scCntRunTimeReg()
setControl()
showLeds()
statReg()
```

Concurrency: Guarded

Persistence: Persistent

5.7.1 clrControl()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears bits, designated by 1's in the *mask* argument, in the BEP Control Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.2 cmdFifoReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned short***

Documentation:

This function returns a pointer to the BEP's Command FIFO. The FIFO is 16-bits wide. 32-bit fetches yield the next value in the FIFO in the least-significant 16-bits and garbage in the upper 16-bits. The pointer itself never changes. The "const" directive indicates that the register cannot be modified by the software. Since the hardware changes the contents of the register with each read, a "volatile" directive is needed to ensure that the compiler's code-optimizer does not remove code which reads from the FIFO (for example, by caching the read value in a CPU register).

Concurrency: Synchronous

5.7.3 ctlReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Control Register.

Concurrency: Synchronous

5.7.4 dtcAddrCntReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller Address Count Register.

Concurrency: Synchronous

5.7.5 dtcEndReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller End Address Register.

Concurrency: Synchronous

5.7.6 dtcStartReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Downlink Controller Start Address Register.

Concurrency: Synchronous

5.7.7 deaCmdReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Detector Electronics Assembly Command Register.

Concurrency: Synchronous

5.7.8 deaStatReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned***

Documentation:

This function returns the constant address of the read-only (“const” directive) BEP Detector Electronics Assembly Status Register. The value of this register can change without being written to, hence the “volatile” directive.

Concurrency: Synchronous

5.7.9 getControl()

Public member of: **BepReg**

Return Class: **unsigned**

Documentation:

This function returns the current value contained in the BEP's Control Register.

Concurrency: Synchronous

5.7.10 getStatus()

Public member of: **BepReg**

Return Class: **unsigned**

Documentation:

This function returns the current value contained in the BEP's Status Register.

Concurrency: Synchronous

5.7.11 pulseReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the BEP Pulse Register.

Concurrency: Synchronous

5.7.12 pulse()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function writes *mask* into the BEP's Pulse Register. This has the effect of pulsing the bits indicated by 1's in the *mask* argument. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.13 scCntLatTimeReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the S/C Latched Counter.

Concurrency: Synchronous

5.7.14 scCntRunTimeReg()

Public member of: **BepReg**

Return Class: **unsigned***

Documentation:

This function returns the constant address of the S/C Running Counter.

Concurrency: Synchronous

5.7.15 setControl()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets bits, designated by 1's in the *mask* argument, in the BEP Control Register. Bits designated by 0's in the *mask* are unaffected.

Concurrency: Synchronous

5.7.16 showLeds()

Public member of: **BepReg**

Return Class: **void**

Arguments:

unsigned *value*

Documentation:

This function sets the LED bits in the BEP's Control Register to the argument *value*. The least-significant four bits of *value* correspond to the four LED bits in the BEP Control Register. All other bits in *value* are ignored. Only the LED bits in the Control Register are modified by this function.

Concurrency: Synchronous

5.7.17 statReg()

Public member of: **BepReg**

Return Class: **volatile const unsigned***

Documentation:

This function returns the constant address of the read-only (“const” directive) BEP Status Register. The value of this register can change without being written to, hence the “volatile” directive.

Concurrency: Synchronous

5.8 Class Leds

Documentation:

This class is responsible for writing LED values to the software discrete telemetry bi-levels.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg

Public Interface:

Operations: `show()`

Concurrency: Synchronous

Persistence: Transient

5.8.1 show()

Public member of: **Leds**

Return Class: **void**

Arguments:

unsigned *value*

Documentation:

This function writes value to the LED discrete telemetry bi-levels, by passing *value* to *bepReg*.`showLeds`(). *value* must range from 0 to 15, inclusive.

Concurrency: Synchronous

5.9 Class BootMode

Documentation:

This class is responsible for determining the type of boot the BEP came up from.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg

Public Interface:

Operations: `isPowerOn()`
`isWatchdog()`

Concurrency: Sequential

Persistence: Transient

5.9.1 isPowerOn()

Public member of: **BootMode**

Return Class: **Boolean**

Documentation:

This function determines if the BEP was reset due to a power-on reset. If so, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Semantics:

Call *bepReg*.getStatus() and test for a commanded or watchdog reset. If neither is set, then the BEP came up due to a power-on, and return *BoolTrue*. If either the Command Reset or Watchdog Reset status bits are set, then return *BoolFalse*.

Concurrency: Synchronous

5.9.2 isWatchdog()

Public member of: **BootMode**

Return Class: **Boolean**

Documentation:

This function determines if the BEP was last reset due to the watchdog timer. It returns *BoolTrue* if so, and *BoolFalse* if the watchdog did not cause the last reset.

Semantics:

Call *bepReg*.getStatus() and test the Watchdog reset status bit. If set, then the BEP was reset by the Watchdog timer. Return *BoolTrue*. If the Watchdog reset status bit is not set, then return *BoolFalse*.

Concurrency: Synchronous

6.0 Interrupt Control Classes (36-53206 A)

6.1 Purpose

The purpose of the collection of interrupt control classes is to manage interrupts from the various Back End Processor devices. For a description of the R3000 and Mongoose interrupt structure and registers, see Section 4.1 and Section 4.2 . For a list of Back End Processor interrupt causes, priorities and their timing requirements, see Section 4.3.13 .

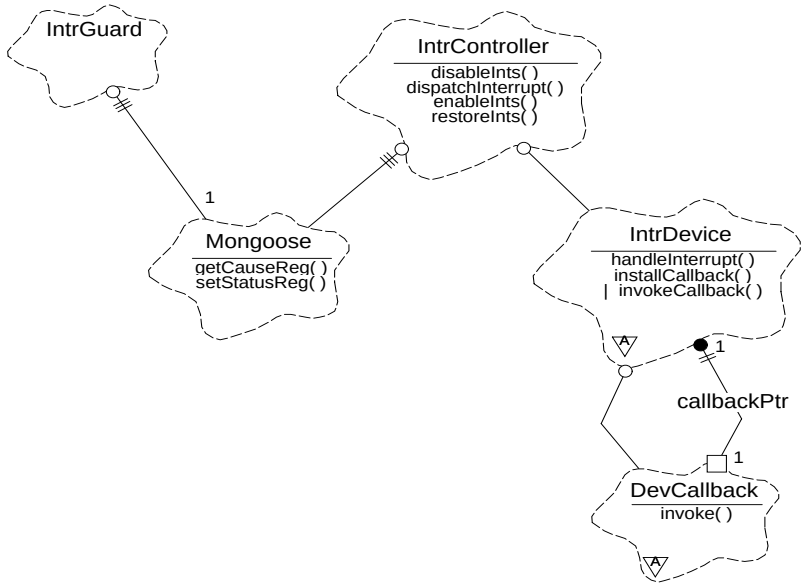
6.2 Uses

Use 1:: Dispatch control to a device when the associated hardware interrupt is asserted
Use 2:: Provide the ability to disable interrupts when executing critical sections of code
Use 3:: Allow clients of interruptible devices to obtain control during interrupt processing

6.3 Organization

This collection of classes consists of (a) an **`IntrController`** class, which manages prioritized interrupts and dispatches control to the interrupting device, (b) an **`IntrGuard`** class, whose construction and destruction delimit periods during which interrupts are disabled, (c) an abstract **`IntrDevice`** class, which generalizes the common interface to all interrupting devices, and (d) an abstract **`DevCallback`** class, which provides the common interface to all classes which can be installed to be “called-back” during interrupt processing. The **`IntrController`** class is a top level class, and uses the **`Mongoose`** class to provide access to the R3000 and Mongoose interrupt control and status registers. Not shown in the figure is the low-level R3000 assembly language code which passes control to the **`IntrController`** during interrupt processing.

FIGURE 22. Interrupt Controller and Device Relationships



`IntrController` - The **`IntrController`** class is responsible for top-level interrupt handling (`dispatchInterrupt`), and for enabling (`enableInts`), disabling (`disableInts`) interrupts, and for restoring a previous interrupt-enable state (`restoreInts`). It uses the **`Mongoose`** class to manipulate the R3000 Co-processor 0's Status Register (`setStatusReg`), read Co-processor 0's Cause Register (`getCauseReg`), and to read and write to the Mongoose Control/Status Interface's (CSI) Extended Mask and Cause registers (*`mongoose.regs->xmask`* and *`mongoose.regs->xcause`*, not shown in Figure 22).

`IntrDevice` - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the **`IntrController`** to dispatch control to interruptible devices (`handleInterrupt`()), and by client code to install callback functions (`installCallback`), which are invoked during a devices's interrupt processing (`invokeCallback`() and **`DevCallback`**::`invoke`()). Child classes of **`IntrDevice`**, may use their parent's protected method, `invokeCallback`() to invoke the installed callback instance.

`DevCallback`- This is an abstract class which defines the common interface to all classes which can be installed as a device callback (via **`IntrDevice`**::`installCallback`()). The **`IntrDevice`** class uses the **`DevCallback`** member function `invoke`() to pass control to the callback instance during the device's interrupt processing.

`IntrGuard` - This class provides a "safe" mechanism by which client code can temporarily disable interrupts during critical sections of code. The constructor for **`IntrGuard`** uses **`Mongoose`**::`setStatusReg`() to read the current interrupt enable state and disable interrupts. The read state is stored in a private variable within the **`IntrGuard`** instance. When the destructor for the class is invoked, it uses the same **`Mongoose`** function to restore the saved interrupt enable state. By declaring an instance of **`IntrGuard`** within a

local scope (i.e. a function or sub-block within a function) interrupts are disabled from the point of declaration until the code leaves the scope of the block. By relying on the compiler to generate the destructor when a function returns or a local block is exited, it ensures that the original interrupt state is restored.

`Mongoose` - This class is used by the **`IntrController`** class to manage the R3000 Coprocessor 0's Status and Cause registers, and the Mongoose Extended Interrupt Mask and Cause registers. Refer to Section 5.0 for a description of this class.

6.4 R3000 and Mongoose Interrupt Description

6.4.1 R3000 Interrupt Status and Cause Registers

The R3000 manages interrupts through its System Co-processor (Co-processor 0). This co-processor contains a Status Register and Cause Register (see Section 4.1). The Status Register controls interrupt enables and disables, and contains (in addition to many other control bits) 8 interrupt mask bits. Two of these bits correspond to the two R3000 software interrupts, and the other 6 correspond to hardware interrupts level-triggered lines wired to the R3000. In addition to various other status information, the co-processor's Cause register contains an Exception cause code and 8 interrupt cause bits, corresponding to the mask bits in the Status Register.

The ACIS instrument software only deals with the Interrupt Exception Cause. All other Exception codes are considered fatal errors, and during pre-flight debugging, will cause a fatal system error and reboot.

6.4.2 Mongoose Extended Interrupt Mask and Cause Registers

In addition to the R3000 interrupts, the Mongoose Microcontroller adds another suite of interrupt causes, including a Watchdog Timer Interrupt, General Purpose Timer Interrupt, DMA Interrupt, UART Receive Interrupt, UART Transmit Interrupt and three external edge triggered interrupts, and a collection of additional error exceptions (see Section 4.2). These interrupts are ORed to the R3000 hardware interrupt number 3. The device interrupts (DMA, Timers, and UART) can be individually masked via a memory-mapped register in the Mongoose's CSI block, the Extended Interrupt Mask register. The cause of a particular Mongoose Interrupt is indicated by the Extended Cause register, in the same CSI block.

6.4.3 Low-level Interrupt Processing

When the R3000 is interrupted, it saves the interrupt return address in its Co-processor 0's Exception Program Counter register, disables interrupts, places the processor into "Kernel" mode and executes its Exception Vector code located at processor address 0x80000080. This code branches to the main low-level interrupt handler, written in assembler. This handler then saves a few registers, invokes the ***Nucleus RTX*** function `SKD_Interrupt_Context_Save`(), which saves all of the R3000 registers, including the R3000 Co-processor's Exception Program Counter and Status Register, and some information on the task being interrupted. The assembler handler then calls a "C/C++" function, `intr_handler`(). This function then invokes the **`IntrController`**::`dispatchInterrupt`() function. Once **`IntrController`** and `intr_handler`() return, the assembly handler calls `SKD_Interrupt_Context_Restore`() to restore control to a task.

6.4.4 Interrupt Priorities

The R3000 does not provide built-in facilities for nesting prioritized interrupts. It treats this as a software responsibility. The recommended technique (see “The MIPS Programmer’s Handbook,” Section 4) for implementing prioritized nested interrupts uses a 256 entry table, indexed by the 8 interrupt cause bits from the Co-processor’s Cause register. During initialization, the software sets up the table to map the 8 possible causes to a single priority selection. A second table, indexed by priority, selects an interrupt mask to use. During interrupt processing, while interrupts are disabled, the software uses the Cause bits to select the priority, and then uses priority to select a mask which disables all lower-priority interrupt causes. It then writes the mask to the Co-processor’s Status Register, and re-enables interrupts. This will allow higher priority interrupts to occur while the software is processing the current interrupt level.

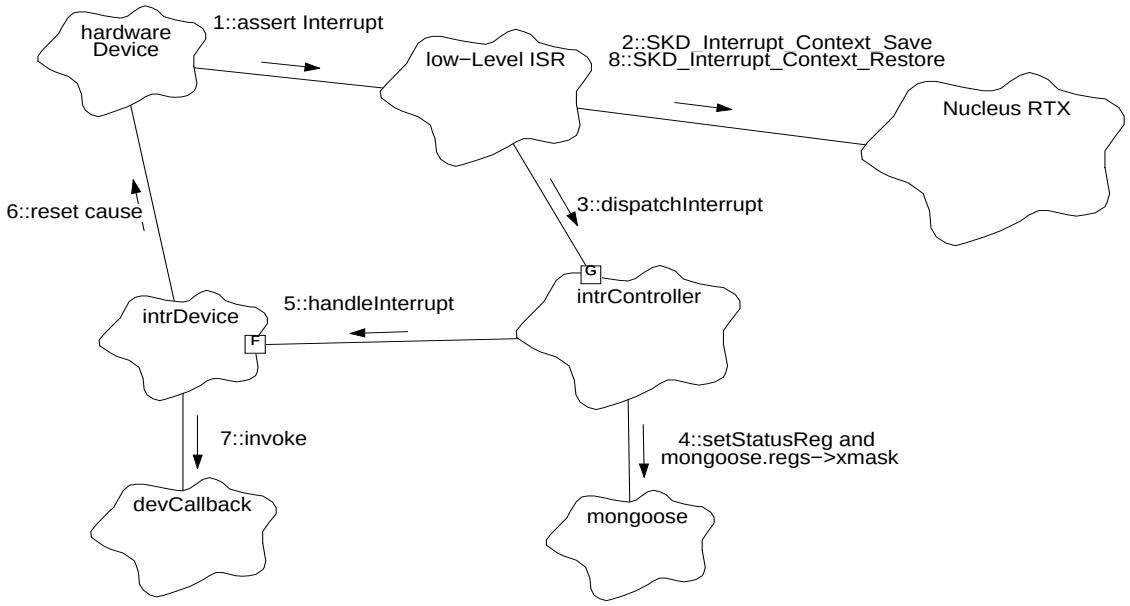
Given that the Mongoose provides another set of interrupts (see Section 4.2), located in a different register, and masked using a different mask, it is difficult for the ACIS software to use exactly the same scheme for prioritizing interrupts. Instead, the **`IntrController`**::`dispatchInterrupt`() function tests for each of the known interrupt causes separately, and selects both an R3000 mask and Mongoose Extended Interrupt mask which disables lower-priority device interrupts. The function then masks both the R3000 and Mongoose causes, enables interrupts, and transfers control to the interrupt device, using **`IntrDevice`**::`handleInterrupt`(). Once the device returns, the dispatch function disables interrupts and restores the previous R3000 and Mongoose interrupt masks. For a list of Back End hardware interrupts and priorities, see Section 4.3.13 .

6.5 Scenarios

6.5.1 Use 1::Dispatch Control to Device on Hardware Interrupt

Figure 23 illustrates the overall interrupt handling scenario.

FIGURE 23. Interrupt Handling Scenario



1. A hardware device asserts its interrupt line, causing the R3000 to execute its Exception Vector code, which then branches to the instrument software's low-level Interrupt Service Routine (ISR).
2. The low-level Interrupt Service Routine calls ***Nucleus*** to save the register and task context (`SKD_Interrupt_Context_Save`())
3. The low-level ISR then invokes `intr_handler`() (not shown) which in-turn invokes *`intrController`*.`dispatchInterrupt`().
4. `dispatchInterrupt`() gets the R3000 Co-processor 0's Cause Register and Mongoose Extended Cause (not shown) and then tests each cause in order of priority. Once the cause is determined, it selects the corresponding interrupt device instance to invoke, and determines which higher priority interrupts to enable. It then sets the Mongoose Extended Interrupt Mask register and R3000 Status register to mask off this interrupt cause and all lower priority interrupt causes. It then re-enables interrupts.
5. `dispatchInterrupt` () then invokes the selected device's **`IntrDevice`**::`handleInterrupt`() member function. (NOTE: Each subclass of **`IntrDevice`** is required to provide its own implementation of this function).
6. The member function then resets the cause of the interrupt in the hardware device, and performs any low-level hardware-specific maintenance operations.

7. The member function then invokes the installed callback instances **`DevCallback::invoke`**() function to handle any higher level device-specific operations. (NOTE:: Each subclass of **`DevCallback`** is required to implement its own version of this `invoke`()).
8. Once all interrupt processing is complete (i.e. **`DevCallback`**::`invoke`(), **`IntrDevice`**::`handleInterrupt`(), **`IntrController`**::`dispatchInterrupt`() return, the low-level ISR calls ***`Nucleus RTX`*** `SKD_Interrupt_Context_Restore`() to restore the context of the next task to run (or the same task if no context switch occurred as a result of the interrupt).

6.5.2 Use 2:: Provide Interrupt Disable/Restore for Critical Code Sections

In order to allow client code to perform a set of operations without worrying about one or more of the interrupt handlers modifying things behind its back, the system provides an **`IntrGuard`** class provides the ability to temporarily disable interrupts. This system relies on the C++ behavior of invoking a class constructor when an instance of that class is declared within a function or block within a function, and invoking its destructor when the function returns, or the block is ended.

The following C++ code fragment illustrates how to use the **`IntrGuard`** within a function.

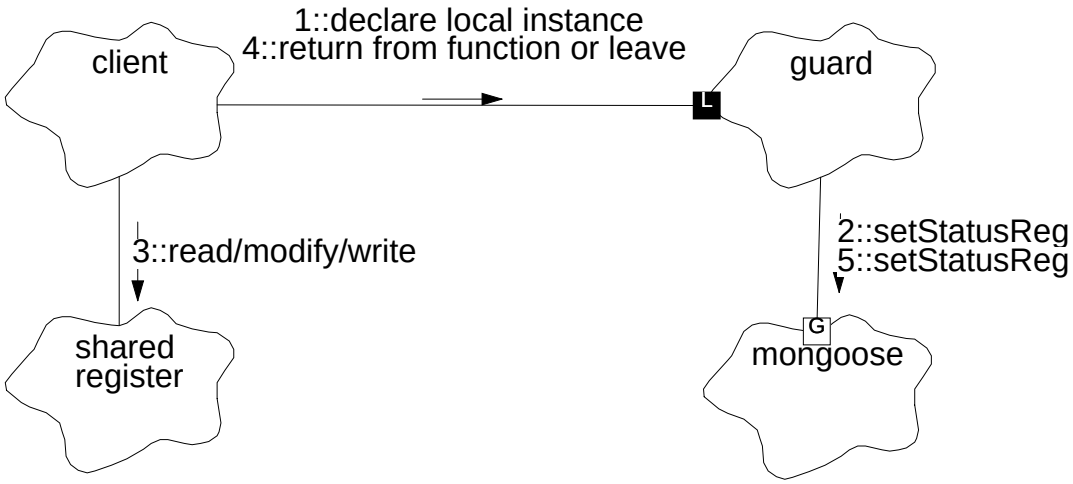
```
// ---- Variable shared by both foo() and an interrupt handler ----
volatile unsigned sharedVariable;

void foo ()
{
    // By declaring guard, interrupt state is saved,  and interrupts are disabled
    // (The private, const variable guard.oldState contains the previous
    // interrupt enable state)
    IntrGuard guard;

    // --- Read variable, increment and write back
    sharedVariable++;
    if (sharedVariable < 10)
    {
         return;// -- Interrupt restored by guard destructor
    }
    sharedVariable = 0;
    return;      // --- Interrupt state restore by guard destructor
}
```

Another example shows its use within a block inside a function:

```
// ---- Variable shared by both bar() and an interrupt handler ----
volatile unsigned sharedVariable;
unsigned nonSharedVariable;

void bar()
{
    nonSharedVariable++;
    {
        // Constructor saves old state and disable interrupts
        IntrGuard guard;
        sharedVariable++;// Safely read/modify/write
    } // Leaving block invokes destructor and restores interrupts

    // Modifying sharedVariable outside block may cause problems
    // due to contention with interrupt handler.
}
```

Figure 24 illustrates the overall use of **`IntrGuard`**.

FIGURE 24. Performing block interrupt disables and restores



1. Client code declares *guard*, a local instance of an **`IntrGuard`**, causing its constructor to be invoked (**`IntrGuard`**::`IntrGuard`()).
2. The constructor, **`IntrGuard`**::`IntrGuard`() invokes **`Mongoose`**::`setStatusReg`() to get the old contents of the R3000 Co-processor 0's Status Register, and to write an interrupt disable value to the register.disable interrupts. **`IntrGuard`**::`IntrGuard`() uses the returned value to initialize its const (read-only) *`oldState`* instance variable.
3. The client code modifies shared variable or register at will, without worrying about interrupt handlers modifying the same variable/register.

4. Upon completion of the protected code, the client code either returns from the protected function, or leaves the protected block of code. Upon leaving a function or block which declared the **`IntrGuard`** instance, the guard's destructor is invoked (**`Intr-Guard`**::~`IntrGuard`()).
5. The guard's destructor, ~`IntrGuard`() passes the saved interrupt enable state, oldState, to **`Mongoose`**::`setStatusReg`() which restores the contents of the R3000 Coprocessor 0's Status Register.

6.5.3 Use 2:: Pass Control to Client Code during Interrupt Processing

In order to support higher level operations during interrupt processing, the **`IntrDevice`** class contains a pointer to an installed **`DevCallback`** instance. Client applications install the callback during initialization using **`IntrDevice`**::`installCallback`(). When an interrupt handler is invoked, the handler is responsible for invoking the callback, **`DevCallback`**::`invoke`(). **`IntrDevice`** provides a function which does this, **`IntrDevice`**::`invokeCallback`(), which in general covers most needs. Figure 22, "Interrupt Controller and Device Relationships," on page 178 illustrates how and when the callback is invoked.

6.6 Class IntrDevice

Documentation:

The **`IntrDevice`** class is an abstraction of all interruptible devices. It serves to provide a set of common interfaces to all such devices. All sub-classes of this class MUST implement the own version of the following: `handleInterrupt` ()

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

`DevCallback`

Public Interface:

Operations: `handleInterrupt()`
`installCallback()`

Protected Interface:

Operations: `invokeCallback()`

Private Interface:

Has-A Relationships:

`DevCallback*` *`callbackPtr`*: This is a pointer to an instance of **`DevCallback`**. This is used by clients to obtain control whenever a device interrupt occurs.

Concurrency: Synchronous

Persistence: Persistent

6.6.1 handleInterrupt()

Public member of: **`IntrDevice`**

Return Class: **`void`**

Documentation:

This member function handles any device-specific interrupt operations. This function MUST be implemented by all leaf subclasses of **`IntrDevice`**.

Semantics:

Handle the device specific interrupt. Usually, subclass implementations of this function reset the device-specific cause of the interrupt and then invoke **`IntrDevice`**::`invokeCallback`()

Time complexity: < 10ms

Concurrency: Synchronous

6.6.2 installCallback()

Public member of: **`IntrDevice`**

Return Class: **`void`**

Arguments:

`DevCallback*` *`callback`*

Documentation:

This function is used by a client to install an **`DevCallback`** instance, pointed to by *`callback`*, to be invoked whenever a device interrupt occurs.

Preconditions:

Another *`callback`* must not have been previously installed (i.e. *`callbackPtr`* must be NULL)

Semantics:

Store *`callback`* in *`callbackPtr`*.

Concurrency: Sequential

6.6.3 invokeCallback()

Protected member of: **`IntrDevice`**

Return Class: **`void`**

Documentation:

This function invokes the installed callback function: **`DevCallback`**::`invoke`(). If no callback is installed, this function has no effect.

Semantics:

If no callback installed, do nothing, otherwise, invoke the installed callback, *`callbackPtr`*->`invoke`().

Time complexity: < 10ms

Concurrency: Synchronous

6.7 Class DevCallback

Documentation:

The **`DevCallback`** class is an abstract class which defines a common interface to all callback functions invoked by device-specific interrupt handlers. All subclasses of this class MUST implement their own versions of the following functions: `invoke`(**`IntrDevice`***)

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations: `invoke()`

Concurrency: Synchronous

Persistence: Persistent

6.7.1 invoke()

Public member of: **`DevCallback`**

Return Class: **`void`**

Arguments:

`IntrDevice*` *`device`*

Documentation:

This function is invoked by a device-specific interrupt handler. This function then performs whatever client specific operations are needed. It is intended that this function be implemented by the various sub-classes of this class.

Semantics:

Perform client-specific device interrupt operations

Concurrency: Synchronous

6.8 Class IntrController

Documentation:

`IntrController` is responsible for dispatching control to various devices when an interrupt occurs.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`none`**

Public Uses:

`IntrDevice`

Implementation Uses:

`Mongoose`
`Dma`
`Watchdog`
`Timer`
`CmdDevice`
`TlmDevice`

Public Interface:

Operations:

```
disableInts()
dispatchInterrupt()
enableInts()
restoreInts()
```

Concurrency: Synchronous

Persistence: Persistent

6.8.1 disableInts()

Public member of: **`IntrController`**

Return Class: **`unsigned`**

Documentation:

Shutdown all interrupts, and return previous interrupt state.

Semantics:

Use *`mongoose`*`.setStatusReg`() to disable interrupts while obtaining the previous interrupt state.

Concurrency: Synchronous

6.8.2 dispatchInterrupt()

Public member of: **`IntrController`**

Return Class: **`void`**

Arguments:

`unsigned` *`icause`*
`unsigned` *`xcause`*

Documentation:

This function dispatches control to each device with a pending interrupt. Each device is serviced in prioritized order, with higher priority devices having the interrupts re-enabled.

Semantics:

Use *`icause`* and *`xcause`* to determine which device is requesting service. While interrupts are disabled, unmask all higher priority interrupt causes on both the R3000 and the Mongoose XMask and re-enable interrupts. Then invoke the interrupting device `handleInterrupt`() function. Once the function returns, disable interrupts and restore the previous Mongoose and R3000 interrupt mask states. Then re-read the R3000 and Mongoose interrupt cause registers, and handle the next cause (prevents unnecessary context saves and restores).

Time complexity: < 10ms

Concurrency: Synchronous

6.8.3 enableInts()

Public member of: **`IntrController`**

Return Class: **`unsigned`**

Documentation:

This function enables interrupts. It returns the previous interrupt enable/disable state.

Semantics:

Use *`mongoose`*`.setStatusReg`() to enable interrupts while retrieving the previous interrupt state.

Concurrency: Synchronous

6.8.4 restoreInts()

Public member of: **`IntrController`**

Return Class: **`void`**

Arguments:

`unsigned` *`oldState`*

Documentation:

Restore a previous interrupt enable/disable state. *`oldState`* is the value returned from `disableInts`() or `enableInts`().

Semantics:

Use *`mongoose`*`.setStatusReg`() to restore the passed interrupt state.

Concurrency: Synchronous

6.9 Class IntrGuard

Documentation:

This class is used within the scope of a block of code to disable interrupts for the duration of the block. Its constructor saves the current interrupt state and disables interrupts. Its destructor, invoked at the end of a given code block, restores the saved interrupt state.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations: `IntrGuard()`
`~IntrGuard()`

Private Interface:

Has-A Relationships:

`const unsigned` *`oldState`*: This variable is set during the instance's constructor and reflects the interrupt state at the time the guard was created. The variable is used upon destruction to restore the interrupt state.

Concurrency: Synchronous

Persistence: Transient

6.9.1 IntrGuard()

Public member of: **IntrGuard**

Return Class: **IntrGuard**

Documentation:

The constructor for **IntrGuard** disables interrupts while saving the previous interrupt state in *oldState*.

Semantics:

Use *mongoose*.setStatusReg() to get the old interrupt state while disabling interrupt. The previous state is stored by initializing *oldState*. Since *oldState* is read-only, the entire operation of this constructor occurs in the initialization statement portion of the constructor.

Concurrency: Synchronous

6.9.2 ~IntrGuard()

Public member of: **IntrGuard**

Return Class: **void**

Documentation:

The destructor for **IntrGuard** restores the interrupt state prior to when the instance was constructed (see IntrGuard()).

Semantics:

Use *mongoose*.setStatusReg() to restore the state indicated by *oldState*.

Concurrency: Synchronous

7.0 Mongoose Devices (36-53207 A)

7.1 Purpose

The Mongoose devices consists of a Direct-Memory-Access (DMA) Controller, General Purpose Timer, and a Watchdog Timer. The purpose of the Mongoose Device classes are to provide access to the each of these devices. Refer to TBD for a description of the hardware interfaces to these devices.

7.2 Uses

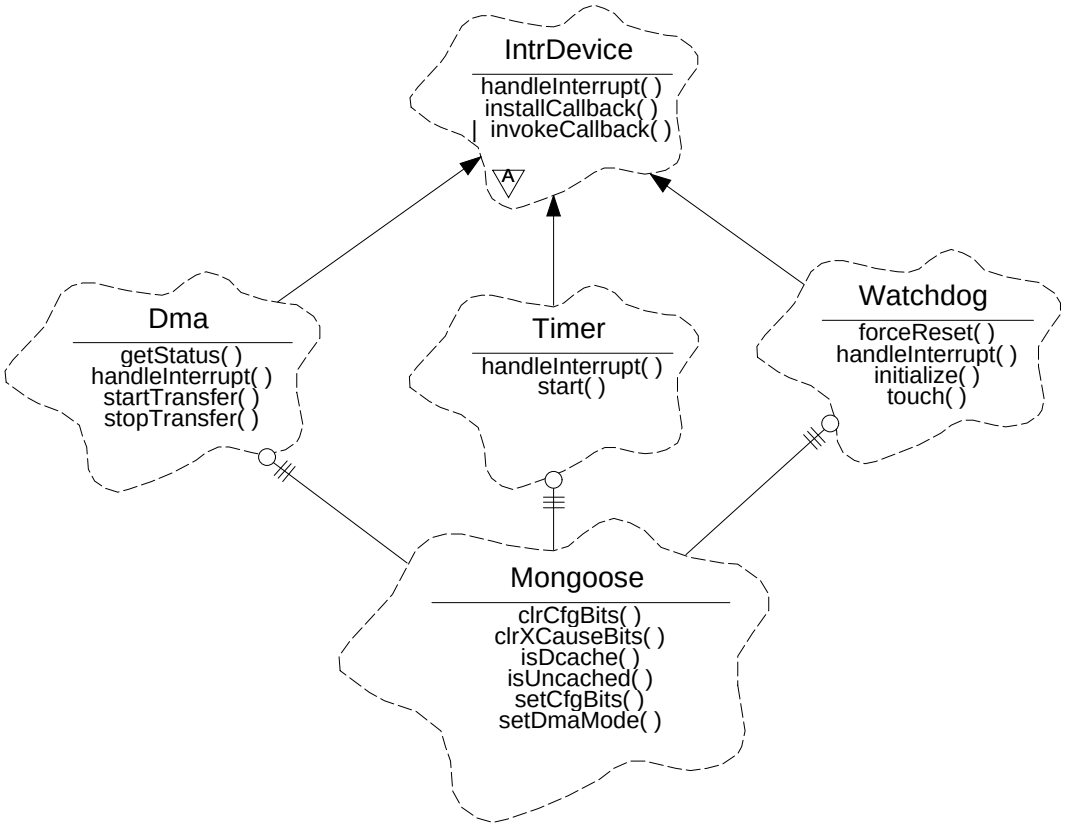
The Mongoose **`Dma`**, **`Timer`** and **`Watchdog`** classes provide the following features:

Use 1:: Initiate DMA Transfers
Use 2:: Stop DMA Transfers
Use 3:: Handle DMA interrupts and forward control to an installed callback object
Use 4:: Manage the General Purpose Timer
Use 5:: Prevent hardware resets by setting Watchdog Timer count
Use 6:: Force a hardware reset using the Watchdog Timer

7.3 Organization

The Mongoose DMA, Timer and Watchdog devices are interruptible, and therefore are all a subclass of **`IntrDevice`** (see Section 6.0). Each of these classes use the **`Mongoose`** class to obtain access to the Mongoose's Command/Status Interface (CSI) registers, and the **`IntrGuard`** class (not shown) to prevent interrupts during certain sections of code. Figure 25 illustrates the relationships used by the **`Dma`**, **`Timer`** and **`Watchdog`** classes.

FIGURE 25. Mongoose Dma, Timer and Watchdog Class Relationships



`Dma` - This class represents the Mongoose DMA device. It provides functions which start and stop DMA transfers (`startTransfer`, `stopTransfer`), return the status of the controller (`getStatus`), and handle interrupts (`handleInterrupt`). In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

`Timer`- This class represents the Mongoose General-Purpose Timer device. It provides functions which start the timer, specifying the interrupt period (`start`), and handle interrupts (`handleInterrupt`). In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

`Watchdog`- This class represents the Mongoose Watchdog Timer device. It provides functions to configure the watchdog time-out period (`initialize`), sets the down-counter to the time-out value (`touch`), and use the timer to force a system (`forceReset`). In order to support debugging when the reset logic is not enabled in the hardware, it also provides a function to handle interrupts (`handleInterrupt`) and inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

`IntrDevice` - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the Interrupt Controller (not shown) to dispatch control to interruptible devices, and by client code to install callback functions. Child classes

of **IntrDevice**, including **Dma**, may use their parent's protected method, invokeCallback() to invoke the installed callback instance (see Section 6.0).

Mongoose - This class represents the lowest level hardware access to the features provided by the R3000 core processor and the Mongoose Microcontroller. The **Dma**, **Timer** and **Watchdog** classes use the **Mongoose** class to obtain access to the Mongoose's DMA, Timer, and Watchdog Count, Timer Count registers, the Mongoose Control Register (via setDmaMode, setCfgBits, and clrCfgBits), and the interrupt cause register (clrXCauseBits). The **Dma** class also uses the Mongoose class to test addresses for legal Mongoose DMA transfers (isDcache, isUncached).

7.4 DMA Transfer Types and Restrictions

The "Mongoose ASIC Microcontroller Programming Guide, Section: D-side Interface Block, DMA Channel" describes the interface to the DMA controller and lists the kinds of transfers supported by the Mongoose's DMA Controller. These transfer types are:

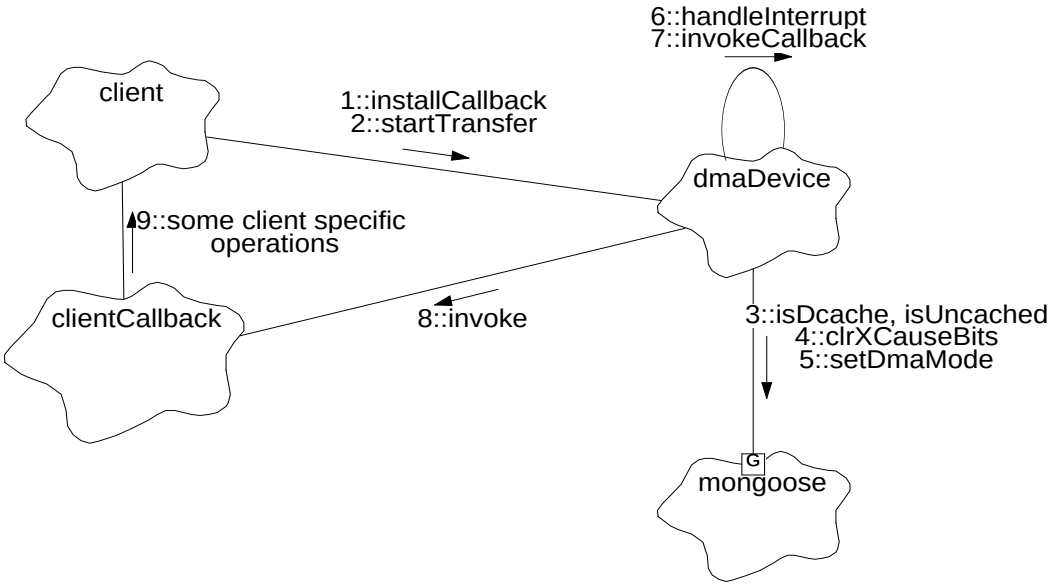
- Transfers from one uncached addressable region to another uncached region
- Transfers from an uncached address region into the data cache
- Transfers from the data cache into an uncached address region

All DMA transfers use 32-bit word reads and writes. When used with a 16-bit memory-mapped device (such as the Command FIFO), the most significant 16-bits of each copied DMA word will contain unspecified values.

7.5 Scenarios

The following diagram is used to illustrate the overall process involved in DMA transfers. Section 7.5.1 and Section 7.5.3 describe the scenarios illustrated by this diagram.

FIGURE 26. DMA Transfers and Interrupt Handling



7.5.1 Use 1: Initiate DMA Transfers

The primary purpose of the DMA controller is to transfer information from one section of memory or device to another. Only one transfer is performed at a time, and it is up to the users of the **`Dma`** class to arbitrate (using `getStatus`) among themselves for access to the controller.

1. During system initialization, the user of the **`Dma`** class constructs a subclass of **`DevCallback`** (not shown) and passes it to the **`Dma`**'s `installCallback`() function (inherited from **`IntrDevice`**).
2. When the client wishes to perform a transfer, it invokes the **`Dma`**'s `startTransfer`() function.
3. `startTransfer`() then determines the type of transfer by inspecting the source and destination addresses, using the **`Mongoose`**'s `isDcache`() and `isUncached`() determine if an address is located within data cache or external memory, respectively.
4. Once the type of transfer is determined, `startTransfer`() uses the **`Mongoose`** function `clrXCauseBits`() to clear any outstanding DMA interrupts.
5. It then uses the **`Mongoose`** "*`regs`*" structure (not shown) to program the source, destination and transfer length registers, and sets the transfer into motion using `setDmaMode`(). Once the transfer completes, a DMA interrupt will be generated and the interrupt controller invokes the **`Dma`**'s `handleInterrupt`() function. The interrupt handling actions are described below in Section 7.5.3

7.5.2 Use 2: Stop DMA Transfers

This use is not illustrated in Figure 26.

To abort a DMA transfer in progress, the user of the **Dma** invokes the `stopTransfer`() function. This function then uses the **Mongoose** functions `setDmaMode` and `clrXCauseBits` to stop the transfer, and clear any pending DMA interrupt.

7.5.3 Use 3: Handle DMA Interrupts

Refer to Figure 26, “DMA Transfers and Interrupt Handling,” on page 200 for an illustration of the scenario described in this section.

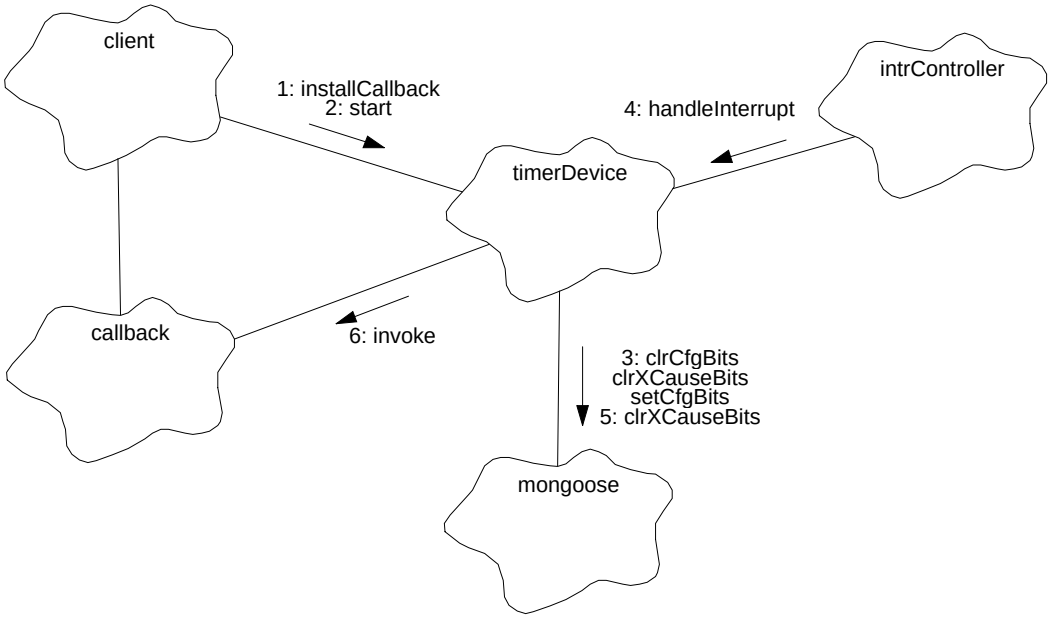
Whenever a DMA interrupt occurs, the Interrupt Controller (not shown) determines the cause of the interrupt, masks off any lower priority interrupt causes and re-enables interrupts. It then dispatches control to the **Dma** class’s `handleInterrupt`(). The following describes the actions taken by the **Dma** class in response to an interrupt.

6. Once the DMA transfer completes, the interrupt controller invokes the **Dma**’s `handleInterrupt`() function. `handleInterrupt`() then clears the interrupt cause latch using the **Mongoose**’s `clrXCauseBits`() function, as in step 4 (see Section 7.5.1).
7. `handleInterrupt`() then invokes **IntrDevice**::`invokeCallback`() (inherited by **Dma** from **IntrDevice**) to test for and invoke the installed callback instance.
8. **IntrDevice**::`invokeCallback`() then calls the installed callback instance’s `invoke`() function.
9. The callback’s `invoke`() function then performs client specific operations needed to deal with the end of the DMA transfer.

7.5.4 Use 4: Manage the General Purpose Timer

Figure 27 illustrates the management of the Mongoose's General Purpose Timer.

FIGURE 27. Timer management

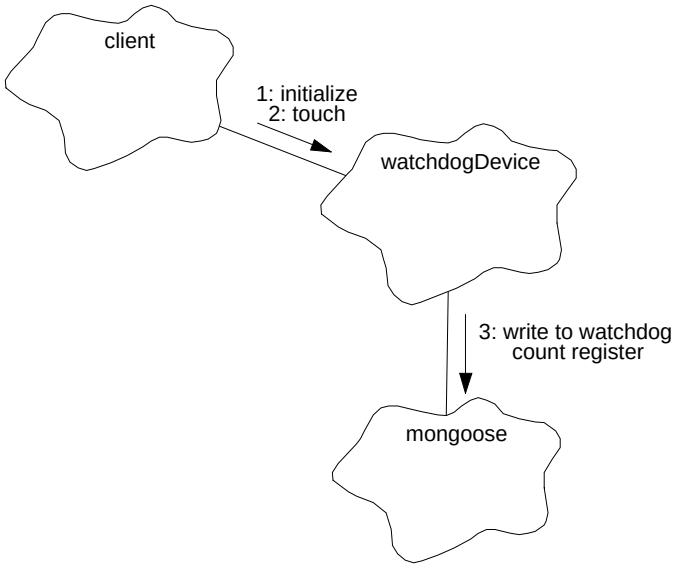


1. A client object installs a callback instance, callback, to be invoked during interrupt processing, using *timerDevice*.installCallback().
2. The client starts the timer, specifying the interrupt period to be generated, using *timerDevice*.start().
3. *timerDevice*.start() saves the passed count into an instance variable, disables the timer enable bit, using *mongoose*.clrCfgBits(), clears any pending timer interrupt using *mongoose*.clrXCauseBits(), writes the passed count into the timer count register (not shown), and enables the timer using *mongoose*.setCfgBits().
4. When the timer's count reaches zero, it generates an interrupt. The interrupt controller object, *intrController*, determines the cause, and tells the device object to handle the interrupt, using *timerDevice*.handleInterrupt().
5. handleInterrupt() re-loads the timer's count register and clears the interrupt using *mongoose*.clrXCauseBits().
6. handleInterrupt() then invokes the callback instance, using *callback*.invoke(). The process repeats from Step 4 until the instrument is reset.

7.5.5 Use 5: Prevent hardware resets by setting Watchdog Timer count

Figure 28 illustrates how a client configures and uses the Watchdog Timer. Since the normal operating environment resets the processor rather than cause watchdog interrupts, the diagram and subsequent description omit interrupt handling.

FIGURE 28. Watchdog Timer Management



1. The client sets the watchdog time-out period using *watchdogDevice*.`initialize`().
2. At a rate within the time-out period, the client prevents a watchdog reset from occurring by periodically calling *watchdogDevice*.`touch`().
3. *watchdogDevice*.`touch`() writes the configured time-out value into the watchdog down-count register. If the client does not call `touch`() within the configured period, the down-count register will reach zero, and the hardware will cause a reset on the Back End Processor.

7.5.6 Use 6: Force a hardware reset using the Watchdog Timer

To force a hardware reset, the client calls *watchdogDevice*.`forceReset`(). `forceReset`() stores a 1 into the Mongoose's watchdog down-count register and attempts to enter an infinite loop. The down-counter decrements once and reaches zero, causing the hardware to generate a physical reset of the Back End Processor.

7.6 Class Dma

Documentation:

This class represents the Mongoose on-chip Direct-Memory-Access (DMA) controller.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`Mongoose`

Public Interface:

Operations:
```
getStatus()
handleInterrupt()
startTransfer()
stopTransfer()
```

Protected Interface:

Has-A Relationships:

`DmaState` *`dmaState`*: This variable contains the current state of the DMA device.

Concurrency: Guarded

7.6.1 getStatus()

Public member of: **Dma**

Return Class: **DmaState**

Documentation:

This function returns the current transfer status of the **Dma** class. The return values are as follows:

DMASTATE_IDLE- No transfer in progress
DMASTATE_BUSY - A DMA transfer is underway

Concurrency: Guarded

7.6.2 handleInterrupt()

Public member of: **Dma**

Return Class: **void**

Documentation:

This function handles DMA interrupts, by clearing the DMA extended cause bit, and by invoking the installed interrupt callback instance.

Postconditions:

If no new transfers are started by the callback instance, subsequent calls to `getStatus`() will return DMASTATE_IDLE. If the callback starts a new transfer, `getStatus`() will return DMASTATE_BUSY if the new transfer is still underway.

Concurrency: Synchronous

7.6.3 startTransfer()

Public member of: **Dma**

Return Class: **Boolean**

Arguments:

```
const unsigned* srcAddress
volatile unsigned* dstAddress
unsigned xfrLength
```

Documentation:

This function initiates a DMA transfer, copying *xfrLength* words from *srcAddress* to *dstAddress*. If the arguments are valid, this function starts the transfer and returns *BoolTrue*. If the arguments are invalid, it returns *BoolFalse*.

Preconditions:

A transfer must not already be in progress.

srcAddress and *dstAddress* and *xfrLength* must be set to support one of the following transfer types:

data cache to external memory/device transfer
external memory/device to data cache transfer
external memory/device to external memory/device.

Semantics:

Test transfer type, clear DMA mode and DMA interrupt, program source, destination and length and set the DMA mode to kick off the transfer

Postconditions:

A transfer will be underway (or already completed if very short). A DMA interrupt may occur after this function returns and the buffer pointed to by *dstAddress* will be changing under one's feet. Calls to getStatus() may return either DMASTATE_IDLE if the transfer has already completed, or DMASTATE_BUSY if the transfer is still underway.

Concurrency: Guarded

7.6.4 stopTransfer()

Public member of: **`Dma`**

Return Class: **`void`**

Documentation:

This function stops a DMA transfer in-progress and clears any pending DMA interrupts.

Semantics:

Set the DMA mode to stop a transfer and clear the DMA Extended Cause bit.

Postconditions:

Any transfers will be aborted. Calls to `getStatus`() will return DMASTATE_IDLE.

Concurrency: Guarded

7.7 Class Timer

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`Mongoose`

Public Interface:

Operations: `handleInterrupt()`
`start()`

Protected Interface:

Has-A Relationships:

`unsigned` *`timeout`*: This is the value to store into the timer's counter upon each interrupt.

Concurrency: Guarded

Persistence: Persistent

7.7.1 handleInterrupt()

Public member of: **`Timer`**

Return Class: **`void`**

Documentation:

This function overloads the **`IntrDevice`** `handleInterrupt` function. For the **`Timer`** device, this function clears the interrupt cause, re-loads the timer's count and invokes the installed callback.

Concurrency: Guarded

7.7.2 start()

Public member of: **`Timer`**

Return Class: **`void`**

Arguments:

`unsigned` *`clockTicks`*

Documentation:

This starts the Mongoose Timer. *`clockTicks`* specifies the number of processor clocks per timeout of the timer.

Concurrency: Guarded

7.8 Class Watchdog

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`Mongoose`

Public Interface:

Has-A Relationships:

`unsigned` *`count`*: This is the time-out count to use. Each call to `touch`() causes this count to be loaded into the timer's counter.

Operations:

```
forceReset()
handleInterrupt()
initialize()
touch()
```

Concurrency: Guarded

Persistence: Persistent

7.8.1 forceReset()

Public member of: **`Watchdog`**

Return Class: **`void`**

Documentation:

This function causes the watchdog timer to reset the instrument hardware by writing a 1 into the timer's down-count register and entering an infinite loop. When the down-counter reaches 0, the hardware will reset the Back End Processor.

Concurrency: Guarded

7.8.2 handleInterrupt()

Public member of: **`Watchdog`**

Return Class: **`void`**

Documentation:

This function overloads the **`IntrDevice`** `handleInterrupt` function, and is used when debugging the instrument software.This function clears the interrupt cause, and invokes the installed callback.

Concurrency: Guarded

7.8.3 initialize()

Public member of: **Watchdog**

Return Class: **void**

Arguments:

unsigned *timeout*

Documentation:

This function sets the time-out period of the watchdog timer. *timeout* is the maximum number of clock cycles that can occur between each call to touch().

Concurrency: Guarded

7.8.4 touch()

Public member of: **Watchdog**

Return Class: **void**

Documentation:

This function copies the configured *count* value into the mongoose's watchdog down-count register. This function must be called at least once every *count* clock cycles, or the timer will expire and reset the processor (or, if debugging, cause a watchdog interrupt).

Concurrency: Synchronous

8.0 Command Device (36-53208 A+)

8.1 Purpose

The purpose of the Command Device is to provide access to Back End's Command Interface logic and to the Command FIFO.

8.2 Uses

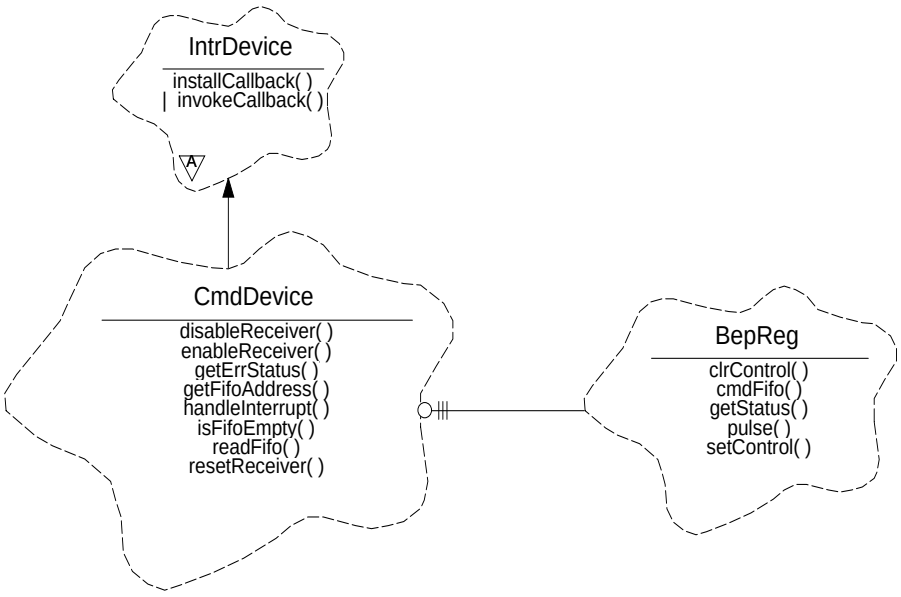
The Command Device class, **`CmdDevice`**, provides the following features:

Use 1:: Disable and reset the command logic and FIFO
Use 2:: Obtain the state of the command FIFO
Use 3:: Enable the command reception logic
Use 4:: Read words from the command FIFO
Use 5:: Obtain the virtual address of the command FIFO
Use 6:: Handle command interrupts, forward control to an installed callback instance.

8.3 Organization

The **`CmdDevice`** is an interruptible device, and is therefore a subclass of **`IntrDevice`** (see Section 6.0). This class relies on the **`BepReg`** class to provide access to the Command hardware control logic and command FIFO.

FIGURE 29. Command Device Class Relationships



`CmdDevice`- This class represents the Back End's Command Device logic and FIFO. It provides functions which control the state of the command reception hardware logic (`disableReceiver`, `enableReceiver`, `resetReceiver`), and provides access to

the Command FIFO (`getFifoAddress`, `isFifoEmpty`, `readFifo`). In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

`IntrDevice` - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the Interrupt Controller (not shown) to dispatch control to interruptible devices, and by client code to install callback functions. Child classes of **`IntrDevice`**, including **`CmdDevice`**, may use their parent's protected method, `invokeCallback`() to invoke the installed callback instance (see Section 6.0).

`BepReg`- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers, and to the Command FIFO. See Section 5.0 for a description of this class.

8.4 Command Logic Description

The Back End's command logic provides the ability to receive entire command packets into the command FIFO and generate an interrupt once the entire packet has been received. This significantly reduces the number of interrupts and the rate of interrupts that the Back End Processor must handle in order to receive a command packet.

Once enabled, the command logic interprets the first received 16-bit command word as the count of the total number of words in the packet, including the received length. It then writes the length into the Command FIFO, and proceeds to receive and store subsequent words from the packet until the last word of the packet has been received. Once the last word is in the FIFO, the hardware generates a command interrupt. The hardware then interprets the next 16-bit words as the length of the subsequent packet.

In the event that the length is out-of-range, or if the command FIFO fills when a packet is being received, the command logic will generate an interrupt, and let the software cope with the condition. The condition is indicated in the Back End's Status register.

Since command transfers and error handling require access to buffer pools and intelligent systems-level decision making, the **`CmdDevice`** class doesn't explicitly handle the transfer of commands from the FIFO into memory, or these error conditions, but instead, provides the low-level access routines used by the higher level protocol class (in this case, the **`CmdManager`**) to deal with these actions.

Refer to Section 4.3.5 for a format description of words read from the Command FIFO. The upper 32-bits of each read word contain status information about the word being read, and the lower 16-bits contain the actual command word received by the DPA hardware from the spacecraft.

8.5 Scenarios

8.5.1 Use 1: Disable and reset command logic and FIFO

In order to support system initialization, and recovery from errors, the **CmdDevice** provides functions which allow the client to disable the command reception logic, disableReceiver(), and to reset the command FIFO, resetReceiver(). The client code uses these routines during initialization and error recovery to stop the hardware logic from generating command interrupts due to erroneous packet lengths, and to quickly discard the contents of the Command FIFO, respectively. In general, the client code should call disableReceiver() prior to resetting the contents of the Command FIFO.

8.5.2 Use 2: Obtain the state of the Command FIFO

In order to allow a client to ensure that the FIFO is empty prior to enabling command reception, or to ensure that the FIFO contains valid data prior to reading and interpreting its contents, the **CmdDevice** provides a function isFifoEmpty() which returns whether or not the FIFO currently contains any data. In order to allow client code to detect errors, it also provides a function, getErrStatus(), which returns the current error status of the Command Logic.

8.5.3 Use 3: Enable command reception

Once the client is ready to receive commands, it uses the **CmdDevice** enableReceiver() function to enable the command reception logic, which in turn will start generating command interrupts upon each reception of a command packet.

8.5.4 Use 4: Read Command FIFO

In order to allow a client to obtain the length of a command packet prior to starting a block transfer from the FIFO, the **CmdDevice** provides a function readFifo() which returns the next word(s) from the Command FIFO.

8.5.5 Use 5: Access Command FIFO Address

In order to allow the client code to perform block transfers from the Command FIFO (possibly by using the Mongoose DMA), the **CmdDevice** provides a function which returns the virtual address of the FIFO, getFifoAddress().

8.5.6 Use 6: Handle command interrupts

Whenever a packet has been placed into the Command FIFO, or an error is detected by the Command Logic, a Command interrupt occurs. Once the Interrupt Controller (not shown in Figure 29) determines that the Command interface caused the interrupt, it masks off any lower priority interrupt causes and re-enables interrupts. It then dispatches control to the

`CmdDevice` class's `handleInterrupt`(). The **`CmdDevice`**'s `handleInterrupt`() function then resets the interrupt cause, and invokes the installed interrupt callback instance. The callback then performs any client specific operations.

8.6 Class CmdDevice

Documentation:

The Command Device is responsible for providing an interface to the ACIS BEP Command hardware.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`BepReg`

Public Interface:

Operations:
```
disableReceiver()
enableReceiver()
getErrStatus()
getFifoAddress()
handleInterrupt()
isFifoEmpty()
readFifo()
resetReceiver()
```

Private Interface:

Has-A Relationships: .

`unsigned` *`intrstatus`*: This is a copy of the Back End Status register, read in the interrupt handler prior to pulsing the clear command interrupt bit. This value is read in the interrupt handler, `handleInterrupt`(), and is zeroed by calls to `disableReceiver`().

Concurrency: Synchronous

Persistence: Persistent

8.6.1 disableReceiver()

Public member of: **`CmdDevice`**

Return Class: **`void`**

Documentation:

This function disables the hardware packet word counter and prevents subsequent “Command Available” interrupts.

Semantics:

Zero *`intrstatus`*, and use **`BepReg`**::`clrControl`() to de-assert the Uplink Enable bit.

Concurrency: Synchronous

8.6.2 enableReceiver()

Public member of: **`CmdDevice`**

Return Class: **`void`**

Documentation:

This function enables the hardware packet word counter, and subsequent “Command Available” interrupts.

Semantics:

Use **`BepReg`**::`setControl`() to assert the Uplink Enable bit.

Concurrency: Synchronous

8.6.3 getErrStatus()

Public member of: **CmdDevice**

Return Class: **enum CmdDevStatus**

Documentation:

Return the current status of the command device. The currently defined values are as follows:

CMDDEV_NOERR - No errors
CMDDEV_ERRLENGTH - Illegal packet length
CMDDEV_ERRFIFOSPILL -Command FIFO filled

Semantics:

Use **BepReg**::`getStatus`() to read contents of status register. If the FIFO Full latch is set, return CMDDEV_ERRFIFOSPILL. If the Command Error bit is set, return CMDDEV_ERRLENGTH, otherwise, return CMDDEV_NOERR.

Concurrency: Synchronous

8.6.4 getFifoAddress()

Public member of: **CmdDevice**

Return Class: **volatile const unsigned***

Documentation:

This function returns a pointer to the Command FIFO. Access to the FIFO address allows the caller to perform block copies or DMA transfers from the FIFO.

Semantics:

Get and return the address of the Command FIFO using **BepReg**::`cmdFifo`().

Concurrency: Synchronous

8.6.5 handleInterrupt()

Public member of: **`CmdDevice`**

Return Class: **`void`**

Documentation:

This function handles an interrupt from the command hardware.

Semantics:

Read the current status register using **`BepReg`**::`getStatus`() and bitwise OR into *`intrstatus`*. Reset the interrupt by pulsing the Clear Uplink Interrupt bit using **`BepReg`**::`pulse`(), and then use **`IntrDevice`**::`invokeCallback`() to pass control to the client code.

Concurrency: Synchronous

8.6.6 isFifoEmpty()

Public member of: **`CmdDevice`**

Return Class: **`Boolean`**

Documentation:

This function determines if the Command FIFO is empty. If so, it returns *`BoolTrue`*, else it returns *`BoolFalse`*.

Semantics:

Get the contents of the BEP's status register using **`BepReg`**::`getStatus`() and test the FIFO empty bit.

Concurrency: Synchronous

8.6.7 readFifo()

Public member of: **CmdDevice**

Return Class: **void**

Arguments:

unsigned* *dst*
unsigned *wordcnt*

Documentation:

This function reads *wordcnt* words from the Command FIFO into the buffer pointed to by *dst*. It is up to the caller to ensure that the FIFO contains at least *wordcnt* words prior to calling this function.

Semantics:

Get the address of the Command FIFO using **BepReg**::cmdFifo(), remove each word from the FIFO and store into *dst*.

Concurrency: Synchronous

8.6.8 resetReceiver()

Public member of: **CmdDevice**

Return Class: **void**

Documentation:

This function resets the command device hardware, including the command FIFO. All data in the FIFO is lost.

Semantics:

Using **BepReg**::pulse(), clear any pending interrupts and errors by asserting the Uplink Interrupt Reset bit in the BEP's Pulse Register and reset the FIFO by pulsing the FIFO reset bit and clear latched error bits. This function clears any errors reported via getErrStatus().

Concurrency: Synchronous

9.0 Telemetry Device (36-53209 A)

9.1 Purpose

The purpose of the Telemetry Device is to provide access to Back End's Telemetry Interface logic and to the Downlink Transfer Controller.

9.2 Uses

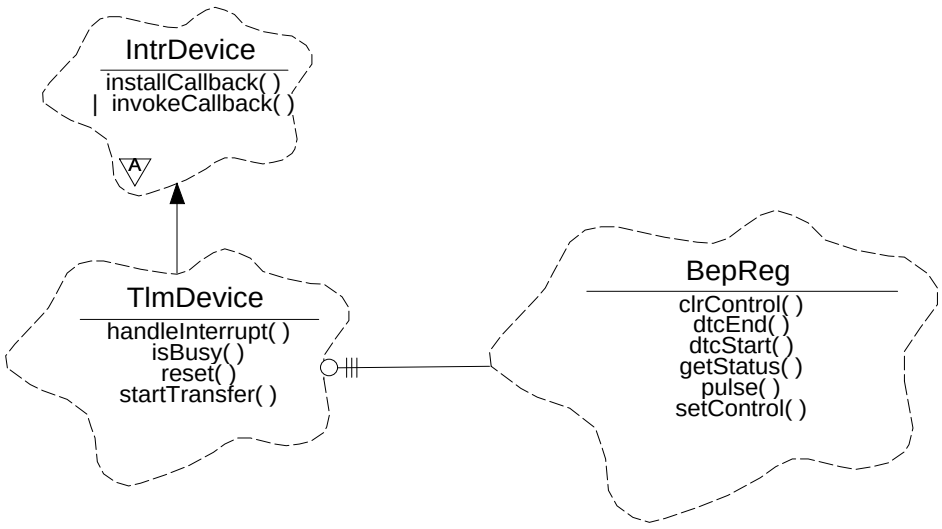
The Telemetry Device class, **`TlmDevice`**, provides the following features:

Use 1:: Initiate a transfer of data from uncached memory to the RCTU hardware
Use 2:: Determine if a telemetry transfer is in progress
Use 3:: Reset the downlink logic
Use 4:: Handle downlink interrupts

9.3 Organization

The Telemetry Device, **`TlmDevice`**, is an interruptible device, and is therefore a subclass of **`IntrDevice`** (see Section 6.0). This class relies on the **`BepReg`** class to provide access to the downlink hardware control logic and Downlink Transfer Controller (DTC).

FIGURE 30. Telemetry Device Class Relationships



`TlmDevice` - This class represents the Back End's downlink logic and controller hardware. It provides functions which reset the controller, start a downlink transfer, determine if a transfer is in-progress, and handle interrupts. In addition to theses functions, it inherits the ability to install and invoke interrupt callback instances from **`IntrDevice`**.

`IntrDevice` - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the Interrupt Controller (not shown) to dispatch control to interruptible devices, and by client code to install callback functions. Child classes of **`IntrDevice`**, including **`TlmDevice`**, may use their parent's protected method, `invokeCallback`() to invoke the installed callback instance (see Section 6.0).

`BepReg`- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers, and to the Downlink Transfer Controller.

9.4 Downlink Transfer Controller Description

The Back End Processor's Downlink Transfer Controller (DTC) is a type of Direct-Memory-Access (DMA) device. It transfers buffers of data from the BEP's uncached memory to the Remote Command and Telemetry Unit (RCTU) telemetry serial port interface logic. This logic handles the insertion of the ACIS time-stamp into the first 32-bits of science data of each Science Telemetry Frame. This logic also provides two deep, 32-bit word FIFO between the DTC and the RCTU. During a transfer, the DTC enqueues 32-bit words from uncached memory into the interface logic's buffer. Once the last word of the block has been transferred from uncached memory, the logic generates a Downlink Interrupt. Assuming that peak transfer rate out of the FIFO is 128Kbps, the 2-deep FIFO gives the software about 0.5 milliseconds to handle the interrupt and start a new transfer before a gap is introduced between the transfers. If a gap is introduced, the hardware will respond to RCTU requests with an 8-bit pattern whose value is 0xb7 in hexadecimal (see Section Section 4.3.6).

9.5 Scenarios

9.5.1 Use 1: Initiate Downlink Transfers from Uncached Memory

To start a telemetry transfer, a client invokes **`TlmDevice`**::`startTransfer`(), passing the start address and number of 32-bit words in the transfer. **`TlmDevice`**::`startTransfer`() then uses **`BepReg`**::`clrControl`() to disable the DTC. It then uses **`BepReg`**::`dtcStart`() and **`BepReg`**::`dtcEnd`() to obtain the address of the start and end registers. It then programs the registers for the transfer, and enables the controller using **`BepReg`**::`setControl`(). Once enabled, the hardware transfers the contents of the buffer to the RCTU interface logic.

It is the callers responsibility to ensure that a transfer is not already in-progress before calling this function.

9.5.2 Use 2: Determine if a transfer is in-progress

A client determines if a transfer is in-progress by calling **`TlmDevice`**::`isBusy`(). Currently, there is no way of determining if a transfer is busy by examining the hardware. In order to support this function, the **`TlmDevice`** class contains a state variable which is asserted whenever a transfer is started, and which is cleared whenever a transfer interrupt is handled. **`TlmDevice`**::`isBusy`() first checks the state variable. If the state variable is de-asserted, then no transfers have been requested, or the previous transfer has completed and its interrupt has been handled. If the state variable is asserted, then either a transfer is underway, or a transfer has completed, but the corresponding interrupt has not yet been handled. In this case, the function uses **`BepReg`**::`getStatus`() to examine the Downlink Interrupt bit. If the bit is de-asserted, then a transfer is still in-progress. If the bit is asserted, then a pending downlink interrupt has not yet been handled, but the previous transfer has completed.

9.5.3 Use 3: Reset the downlink logic

In order to support initialization, and best-effort attempts to send fatal error messages, the **`TlmDevice`** class provides a **`TlmDevice`**::`reset`(). This function ensures that the DTC is disabled, that any pending interrupts are cleared, and places the start and end registers into a known state. Any transfers in-progress when this function is called will be aborted, even though a portion of the previous transfer may have already been sent.

9.5.4 Use 4: Handle downlink interrupts

Whenever all of a request's data have been transferred from uncached memory to the RCTU interface logic, the hardware will generate a Downlink Interrupt. This will eventually cause **`TlmDevice`**::`handleInterrupt`() to be called. This function then clears the transfer busy state variable, resets the interrupt cause, and uses **`IntrDevice`**::`invokeCallback`() to call the telemetry device's callback instance, if one is installed.

9.6 Class TlmDevice

Documentation:

This class provides access to the Back End Processor's telemetry hardware.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`BepReg`

Public Interface:

Operations:
```
handleInterrupt()
isBusy()
reset()
startTransfer()
```

Protected Interface:

Has-A Relationships:

`Boolean` *`busyFlag`*: This flag indicates whether a downlink transfer is underway. If *`BoolTrue`*, a transfer has been started. If *`BoolFalse`*, no transfers are underway.

Concurrency: Synchronous

9.6.1 handleInterrupt()

Public member of: **`TlmDevice`**

Return Class: **`void`**

Documentation:

This function handles the Downlink Controller Interrupt. This function overloads the **`IntrDevice`** function of the same name.

Semantics:

Clear the internal *`busyFlag`*, and then clear the downlink interrupt using **`BepReg`**::`pulse`(). Then invoke **`IntrDevice`**::`invokeCallback`() to pass control to the client code.

Concurrency: Synchronous

9.6.2 isBusy()

Public member of: **TlmDevice**

Return Class: **Boolean**

Documentation:

This function tests to see if a telemetry transfer is underway. If so, it returns *BoolTrue*. If the telemetry hardware is idle, it returns *BoolFalse*.

Semantics:

Test the internal variable *busyFlag*. If cleared, then no transfer is underway. If the flag is set, then use **BepReg**::getStatus() to read the status register. If the downlink interrupt is asserted, then a started transfer just completed, but the interrupt handler hasn't been invoked yet (probably because we're in a higher priority interrupt). Return *BoolFalse*. If the interrupt bit is not asserted, then a transfer is underway and return *BoolTrue*.

Concurrency: Synchronous

9.6.3 reset()

Public member of: **TlmDevice**

Return Class: **void**

Documentation:

This function initializes the Downlink Controller hardware, interrupting any transfer in progress. NOTE: This function should only be used during initialization and during error processing.

Semantics:

Clear the *busyFlag* and then disable the Downlink Controller using **BepReg**::clrControl() and reset the start and end registers to equal values. Clear any pending interrupt using **BepReg**::pulse().

Postconditions:

The downlink controller will be in a idle state.

Concurrency: Sequential

9.6.4 startTransfer()

Public member of: **TlmDevice**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*
unsigned *wordcnt*

Documentation:

This function starts a telemetry transfer of *wordcnt* 32-bit words from *srcaddr* to the RCTU interface hardware.

Preconditions:

A transfer must not already be in progress. *srcaddr* must be in uncached memory and *wordcnt* must be greater than 0. The range *srcaddr* through (*srcaddr* + *wordcnt* - 1) must not cross a 16Kbyte address boundary.

Semantics:

First, set the *busyFlag* to indicate that a transfer is about to start. Then disable the controller using **BepReg**::clrControl(). Use the **BepReg**::dtcStart and **BepReg**::dtcEnd() functions to obtain the start and end register addresses. Program the start and end registers for the transfer and enable the downlink controller using **BepReg**::setControl().

Postconditions:

wordcnt words located at *srcaddr* will be transferred to the RCTU interface hardware. A telemetry interrupt will indicate when the transfer is complete.

Concurrency: Synchronous

10.0 FEP Devices and FEP Interrupt Device (36-53210 A)

10.1 Purpose

The purpose of the Front End Processor (FEP) Device and Front End Processor Interrupt Device classes are to provide access to Back End's Front End Processor Interface logic and interrupt.

10.2 Uses

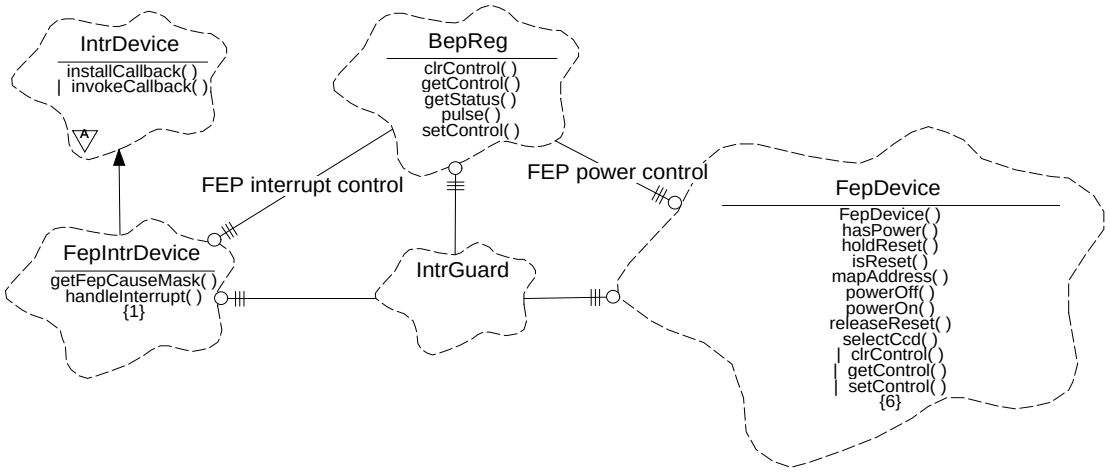
The FEP Device class, **`FepDevice`**, and FEP Interrupt class, **`FepIntrDevice`**, provide the following features:

Use 1:: Handle the interrupt caused by one or more of the active FEPs
Use 2:: Manage the reset lines of each FEP
Use 3:: Map FEP shared memory addresses into BEP address space
Use 4:: Enable and disable power to each of the FEPs
Use 5:: Select which CCD's data is processed by each FEP

10.3 Organization

The Back End Processor manages the Front End Processor hardware using 6 FEP Device class instances, and 1 FEP Interrupt Device class instance. Figure 31 illustrates the relationships used by these classes.

FIGURE 31. FEP Device and Interrupt Device Classes



`FepIntrDevice` - This class is a subclass of **`IntrDevice`** and is responsible for handling interrupts from one or more of the Front End Processors (`handleInterrupt`()), and for providing client access to which Front End Processor's are requesting service (`getFepCauseMask`()). In order to allow client callback functions to process Front End interrupts in tight loops, there is one FEP interrupt (and class

instance) on the Back End Processor. This class uses the **BepReg** class to access the Front End Processor interrupt cause and reset bits (**BepReg**::getStatus(), **BepReg**::pulse()) This class also uses the **IntrGuard** class to temporarily disable interrupts during read/modify/writes of shared registers or instance variables.

FepDevice - This class is responsible for managing the Back End hardware interface logic to Front End Processors. There is one instance of this class for each Front End Processor within ACIS. During start-up, each **FepDevice** instance is associated with one of the Front End Processors by passing to its constructor the corresponding Front End Processor Identifier (**FepId**). From then on, that instance is responsible for managing that FEP's hardware interface (other than interrupts). This class provides functions to map Front End Processor shared memory addresses to the Back End Processor address space (mapAddress), to control and query the state of a Front End Processor's reset line (isReset, holdReset, releaseReset), to query and control the power to a Front End Processor (hasPower, powerOff, powerOn), and to select from which CCD to acquire data (selectCcd). Internally, this class uses the **BepReg** class to gain access to the FEP power control bits (**BegReg**::getControl(), **BepReg**::setControl(), and **BepReg**::clrControl()). This class uses three similar functions, getControl() setControl() and clrControl(), to manage the BEP to FEP Control Register residing on each Front End Processor. It uses **IntrGuard** to disable interrupts when performing read/modify/write operations on these registers.

NOTE: In order to simplify access to these devices, the system provides an array of pointers to these devices, *fepDevice*[], which is indexed by the Front End Processor identifier.

IntrDevice - This is an abstract class which defines the common interface to all types of interruptible devices. It is used by the Interrupt Controller (not shown) to dispatch control to interruptible devices, and by client code to install callback functions. Child classes of **IntrDevice**, including **FepIntrDevice**, may use their parent's protected method, invokeCallback(), to invoke the installed callback instance. See Section 6.0 for more detail.

BepReg- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers. See Section 5.0 for more detail.

IntrGuard - This class is responsible for temporarily disabling interrupts. Its constructor saves the current interrupt state, and disables interrupts. Its destructor restores the previously saved interrupt state. For more detail, see Section 6.0 .

10.4 BEP to FEP Hardware Interface Overview

On ACIS, the active Back End Processor (BEP) is responsible for managing 6 Front End Processors (FEP). Each of these 6 FEPs is responsible for processing science data from one CCD. The following describe certain portions of the Back End/Front End hardware interfaces. For more detail, see Section 4.9 . For details on the software protocols used between the FEPs and the BEP, see Section 4.10 .

10.4.1 Power Control

The Back End Processor is responsible for turning on and off the power to each of the FEPs. The BEP software uses 6 FEP power control bits in the Back End's Control Register, one bit for each FEP. When a bit is set to 1 in the control register, the power to the corresponding FEP is turned on. When the bit is 0, that FEP is powered off.

10.4.2 FEP to BEP Interrupt

Each FEP can cause an interrupt on the Back End Processor. The FEP interrupt lines are combined together to form a single FEP to BEP interrupt. The Back End's Status Register contains 6 bits which each contain a latched version of a given FEP to BEP interrupt line. If any of these bits is set, then a FEP to BEP interrupt is generated on the Back End. The Back End can individually reset each of these latches with a corresponding bit in the Back End's pulse register.

10.4.3 Shared Memory

A portion of each FEP's memory, including their respective reset vectors and microboot control words, appears in the Back End's address space, and can be read from and written to directly by the Back End. When the Back End needs to start a FEP, it holds the FEP's reset line, loads any needed code and data into the FEP via the shared memory interface (including the code at the FEPs reset vector and microboot control word) and releases the FEP's reset line. The FEP will then use the microboot control word to setup its cache address mapping, and start executing the loaded code starting from its reset vector. Once running, the Back End can communicate with each of the FEPs through their respective shared memory (using an agreed upon software interface, see the FEP Manager, Section 25.0 , and the interface descriptions in Section 4.9 and Section 4.10).

10.4.4 FEP Reset Control and Watchdog Timer

Each Front End Processor contains a BEP to FEP control register. This register is writable by the Back End via the shared memory interface, and can only be read by software running on the FEP. This register contains a reset control bit, and a separate reset status bit. The Back End can control the state of the FEP's reset line using the control bit. The current state of the FEP's reset line is indicated by a status bit in this register.

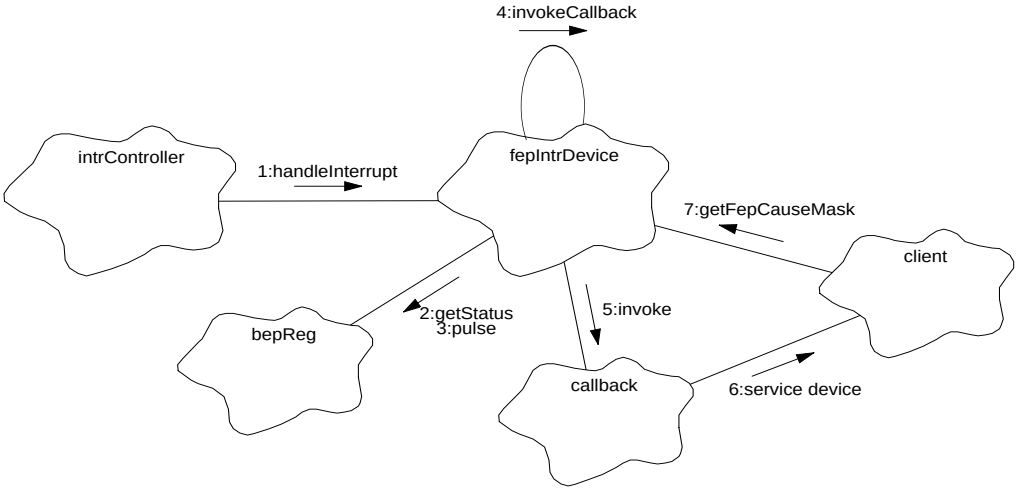
In addition to being controlled by the Back End, a given FEP's reset line is also under the control of the FEPs watchdog timer logic. If a given FEP's watchdog timer expires, the FEP's reset line is asserted and held, and a FEP to BEP interrupt is generated. The FEP's reset line remains asserted until explicitly released by the Back End. This allows the Back End to detect crashes occurring on the FEP and take appropriate action.

10.5 Scenarios

10.5.1 Use1: Handle the interrupt caused by one or more of the active FEPs

Figure 32 illustrates the handling of an FEP interrupt.

FIGURE 32. Handle FEP Interrupt

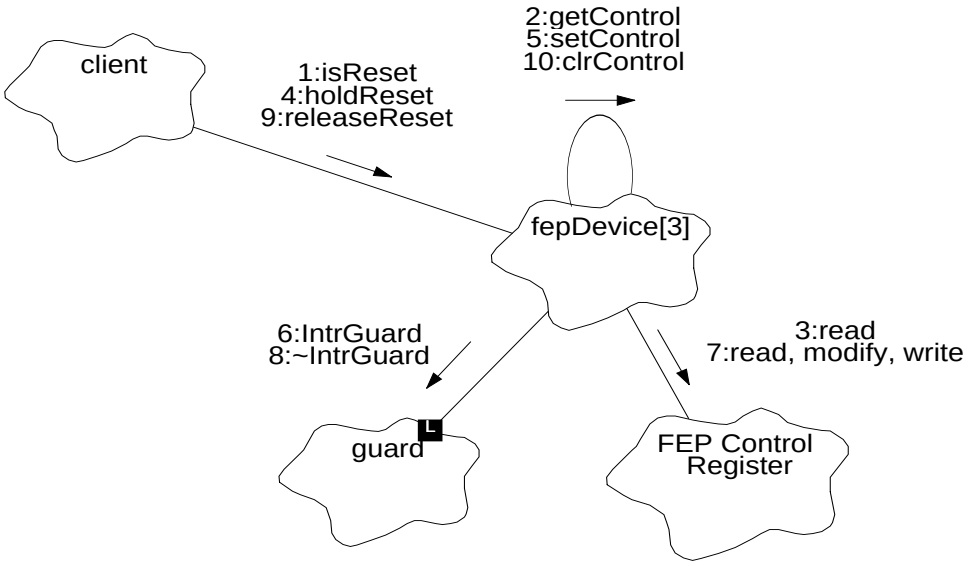


1. One or more of the Front End Processors generate a Back End Interrupt. The interrupt controller instance determines the cause of the interrupt, enables higher priority interrupts (not shown) and invokes the FEP Interrupt Device instance to deal with the interrupt, using *`fepIntrDevice`*.`handleInterrupt`().
2. `handleInterrupt`() uses *`bepReg`*.`getStatus`() to read the Back End's status register. It masks and shifts the returned value to get a bit-field containing which FEPs are currently requesting interrupt service from the BEP. `handleInterrupt`() saves the mask in its private *`saveMask`* instance variable.
3. It then clears the latched FEP interrupt cause bits using *`bepReg`*.`pulse`(). At this point, *`saveMask`* contains the list of FEPs which will be serviced by this interrupt invocation. The handling of subsequent FEP interrupt requests will be deferred until after this handler invocation returns.
4. `handleInterrupt`() calls its inherited function, **`IntrDevice`**::`invokeCallback`() to pass control to the installed FEP Interrupt callback instance.
5. `invokeCallback`() then invokes the installed callback instance's `invoke`() function.
6. *`callback`*.`invoke`() then passes control to a client object (service device).
7. The client then obtains the mask of FEPs requesting service, using *`fepIntrDevice`*.`getFepCauseMask`(), which returns *`saveMask`*, and proceeds to respond to each of the requesting FEP's needs.

10.5.2 Use 2: Manage the reset lines of each FEP

Figure 33 illustrates queries, assertions and de-assertions of an FEP reset line.

FIGURE 33. FEP Reset line query, assertion and de-assertion



1. The *client* queries FEP number 3's reset state, using *fepDevice*[3]->isReset().
2. *fepDevice*[3]->isReset() calls *fepDevice*[3]->getControl() and tests the returned word's reset status bit to determine if the Front Processor is reset. isReset() then returns *BoolTrue* if the processor is in a reset state, or *BoolFalse* if it is not.
3. *fepDevice*[3]->getControl() reads the associated FEP's control register from the shared memory interface.
4. The client calls *fepDevice*[3]->holdReset() to place the FEP into a reset state.
5. *fepDevice*[3]->holdReset() calls its private function setControl() to assert the reset bit in the FEP control register.
6. setControl() first disables interrupts by constructing an **IntrGuard** instance, *guard*.
7. setControl() reads the FEP control register, ORs in the reset bit, and writes it back to the memory-mapped register.
8. setControl() then returns, leaving the scope which created *guard*. *guard*'s destructor (~IntrGuard()) then restores the previous interrupt state.
9. The client calls *fepDevice*[3]->releaseReset() to restart the FEP.
10. releaseReset() calls the internal function, clrControl(). clrControl() then uses **IntrGuard** to disable interrupts, reads the control register, clears its reset bit, writes the result back to the control register, and returns, restoring the previous interrupt state (not illustrated).

10.5.3 Use 3: Map FEP shared memory addresses into BEP address space

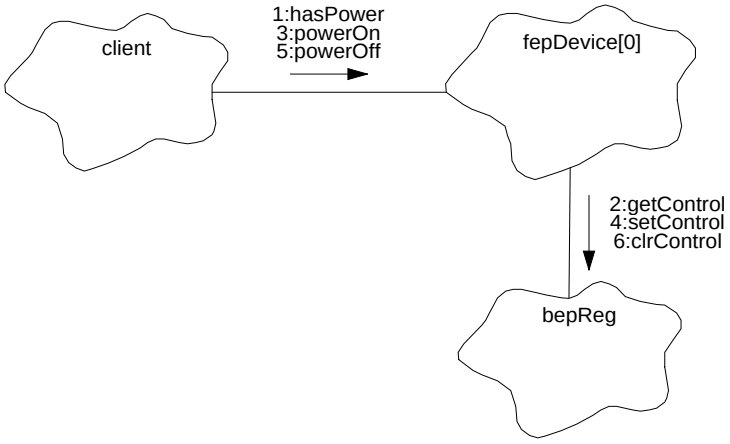
After start-up, when an **`FepDevice`** instance is constructed, it is permanently mapped to a particular Front End Processor. When a client needs to map an address, specified in terms of the Front End Processor's virtual address space, into the corresponding shared memory address in the Back End Processor, it calls the particular FEP's `mapAddress`() member function, passing the starting address and maximum number of words to be accessed. If the specified starting address and range is within a FEP's shared memory space, `mapAddress`() returns the corresponding address in the Back End's virtual address space. If the starting address or ending address is not within the shared memory address space, `mapAddress`() returns 0.

NOTE: The BEP shared memory address space assigned to a particular FEP does not necessarily map to contiguous FEP memory blocks.

10.5.4 Use 4: Enable and disable power to each of the FEPs

Figure 34 illustrates power queries, and power on, and power off commands to a Front End Processor.

FIGURE 34. Querying FEP power, and turning an FEP on and off



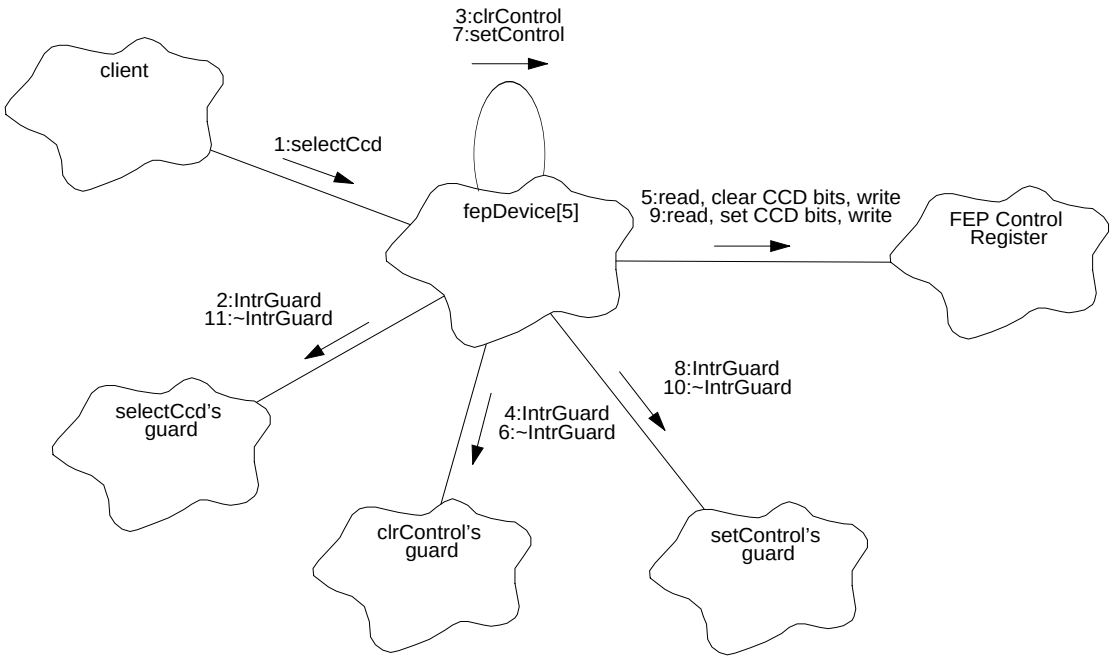
1. The client queries the current state of FEP 0's power, using *`fepDevice`*[0]->`hasPower`().
2. `hasPower`() calls *`bepReg`*.`getControl`() to read the current state of the Back End's Control register. If the power bit corresponding to FEP 0 is set, `hasPower`() returns *`BoolTrue`*, indicating the FEP is on. Otherwise, it returns *`BoolFalse`*, indicating that FEP 0 is currently off.
3. The client calls *`fepDevice`*[0].`powerOn`() to turn on FEP 0.
4. `powerOn`() calls *`bepReg`*.`setControl`() to assert the power control bit for FEP 0, which causes power to be supplied to FEP 0.
5. The client calls *`fepDevice`*[0]->`powerOff`() to turn off power to FEP 0.

6. `powerOff`() calls *`bepReg`*.`clrControl`() to clear the power bit for FEP 0. Once this bit is cleared, FEP 0 is no longer powered on (NOTE: Any state information maintained in FEP 0, including its pixel bias map values, is now lost).

10.5.5 Use 5: Select CCD to process

Figure 35 illustrates how a client configures a FEP to process data from a particular CCD.

FIGURE 35. Assigning CCD Selection



1. The client tells FEP 5 to accept data from CCD I2 (Imaging CCD 2) only, by calling *`fepDevice`*[5].`selectCcd`().
2. `selectCcd`() temporarily disables interrupts by declaring an **`IntrGuard`** instance, *`guard`*.
3. `selectCcd`() then uses `clrControl`() to zero all of the CCD selection bits in the FEP's control register.
4. `clrControl`() declares another local **`IntrGuard`** instance, which saves the already disabled interrupt state.
5. `clrControl`() then reads the FEP Control Register, clears the CCD selection bits, and writes the result back to the control register.
6. `clrControl`() returns, and its local guard's destructor, ~`IntrGuard`(), is invoked. The destructor then restores the disabled interrupt state (i.e. interrupts remain disabled).
7. `selectCcd`() shifts the passed CCD identifier to the appropriate bit position within the control register, and uses `setControl`() to set the 1's contained in CCD selection into the control register (NOTE: `clrControl`() has already cleared the others).

8. `setControl`() declares another local **`IntrGuard`** instance, which also saves the already disabled interrupt state.
9. `setControl`() then reads the FEP Control Register, sets the CCD selection bits, and writes the result back to the control register.
10. `setControl`() returns, causing ~`IntrGuard`() to restore the disabled interrupt state.
11. `selectCcd`() returns, causing its ~`IntrGuard`() to restore the interrupt state that was active when the *`client`* first invoked `selectCcd`().

10.6 Class FepIntrDevice

Documentation:

This class is responsible for handling the FEPs to BEP interrupt. Since any active FEP can cause this single interrupt, **`FepIntrDevice`** provides the capability to query which FEP(s) caused a particular interrupt.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`IntrDevice`**

Implementation Uses:

`BepReg`
`IntrGuard`

Public Interface:

Operations: `getFepCauseMask()`
`handleInterrupt()`

Private Interface:

Has-A Relationships:

`Boolean` *`inHandler`*: This variable indicates whether or not we are currently processing a FEP interrupt. If *`inHandler`* is *`BoolFalse`*, then the device is not inside its `handleInterrupt`() routine. If *`inHandler`* is *`BoolTrue`*, then `handleInterrupt`() has been called and is in the process of dealing with an FEP interrupt.

`unsigned` *`saveMask`*: This contains the FEP interrupt mask, saved prior to resetting the FEP interrupt cause bits by `handleInterrupt`(). This mask indicates which FEPs were requesting service at the time the interrupt handler was invoked (and which cause bits were subsequently cleared by the handler).

Concurrency: Synchronous

Persistence: Persistent

10.6.1 getFepCauseMask()

Public member of: **`FepIntrDevice`**

Return Class: **`unsigned`**

Documentation:

This function returns a value indicating which Front End Processors are requesting interrupt service. The least significant 6-bits in the return value each correspond to one FEP, where bit 0 corresponds to FEP_0, bit 1 to FEP_1 and so on. The remaining bits in the returned value are unused, and will be set to 0. If a particular FEP bit is 1, the corresponding FEP has asserted the FEP to BEP interrupt request. If a bit is 0, the corresponding FEP was not requesting service (or was not handling an FEP interrupt) at the time the interrupt was called.

Semantics:

If in the process of handling an interrupt, *`inHandler`* is *`BoolTrue`*, return the value saved when the handler was invoked, *`saveMask`* (i.e. prior to when the interrupts were reset). If not in interrupt processing, obtain the value directly from the FEP Status register.

Concurrency: Synchronous

10.6.2 handleInterrupt()

Public member of: **`FepIntrDevice`**

Return Class: **`void`**

Documentation:

This function handles interrupts from one or more of the Front End Processors. This function clears the interrupt cause and invokes the installed callback function, using the inherited **`IntrDevice`**::`invokeCallback`().

Semantics:

Get the FEP interrupt cause mask, using *`bepReg`*.`getStatus`(). Shift, mask and store in *`saveMask`*. Set *`inHandler`* to *`BoolTrue`*. Clear the FEP interrupt causes (via *`bepReg`*.`pulse`()). Invoke the callback, `invokeCallback`(). Set *`inHandler`* to *`BoolFalse`* and return.

Concurrency: Synchronous

10.7 Class FepDevice

Documentation:

This class provides an interface to allow the Back End Processor access to the hardware for a single Front End Processor.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg
IntrGuard

Public Interface:

Operations:
`FepDevice()`
`hasPower()`
`holdReset()`
`isReset()`
`mapAddress()`
`powerOff()`
`powerOn()`
`releaseReset()`
`selectCcd()`

Protected Interface:

Has-A Relationships:

`const FepId` *`fepId`*:: This represents which Front End Processor is associated with this instance, set when the instance is constructed.

`volatile unsigned*` *`sharedBase`*::This contains the base address in shared memory for this FEP instance.

Operations:
`clrControl()`
`getControl()`
`setControl()`

Concurrency: Synchronous

10.7.1 FepDevice()

Public member of: **FepDevice**

Arguments:

FepId *fepid*

Documentation:

This function initializes the state of the FEP device and uses *fepid* to relate the constructed instance with a particular physical Front End Processor. It initializes *fepId* to the passed *fepid*. The body of the constructor uses *fepId* to select which shared memory base address to use, and stores the selected base address in *sharedBase*.

Concurrency: Sequential

10.7.2 clrControl()

Protected member of: **FepDevice**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function clears the bits, designated by 1's in the mask argument, in the Front End Processor Control Register associated with this **FepDevice** instance. 0's in *mask* have no effect.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Declare **IntrGuard** instance, *guard*, to save interrupt state and disable interrupts. Read the BEP to FEP control register from shared memory (*sharedBase*[CTL_OFFSET]), bitwise-AND the inverse of *mask* to clear the specified bits, and write the result back out to the control register. Once this function returns, *guard*'s destructor will be invoked, and will restore the previous interrupt state.

Concurrency: Synchronous

10.7.3 getControl()

Protected member of: **FepDevice**

Return Class: **unsigned**

Documentation:

This function returns the current contents of the Front End Processor's (the one associated with this instance) control register.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Read and return the FEP's control register (*sharedBase*[CTL_OFFSET]) from shared memory.

Concurrency: Synchronous

10.7.4 hasPower()

Public member of: **FepDevice**

Return Class: **Boolean**

Documentation:

This function tests to see if the Front End is turned on. It returns *BoolTrue* if so, and *BoolFalse* if the FEP is powered off.

Semantics:

Use *bepReg*.getControl() to read the current value of the BEP control register. Test the bit corresponding to the associated FEP. If 1, return *BoolTrue*, if 0, return *BoolFalse*.

Concurrency: Synchronous

10.7.5 holdReset()

Public member of: **FepDevice**

Return Class: **`void`**

Documentation:

Assert and hold the Front End Processor's reset line, using **`FepDevice`**::`setControl`().

Preconditions:

For sensible results, the FEP must be powered on.

Concurrency: Synchronous

10.7.6 isReset()

Public member of: **`FepDevice`**

Return Class: **`Boolean`**

Documentation:

This function tests the FEP reset line. It returns *`BoolTrue`* if the FEP reset line is asserted, and *`BoolFalse`* if the FEP is running.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Use **`FepDevice`**::`getControl`() to read the FEP's control register, and tests the reset status bit. If 1, return *`BoolTrue`*, else return *`BoolFalse`*.

Concurrency: Sequential

10.7.7 mapAddress()

Public member of: **FepDevice**

Return Class: **volatile unsigned***

Arguments:

volatile const unsigned* *fepaddr*
unsigned *wordcnt*

Documentation:

This function tests to see if the FEP buffer pointed to by *fepaddr* (in FEP address space), and whose size is *wordcnt*, is accessible via the shared-memory interface to the FEP. If so, it returns the corresponding shared memory address. If not, it returns 0.

Semantics:

Verify that *fepaddr* is greater than or equal to the start of a FEP's shared memory address space, and that *fepaddr* + *wordcnt* - 1 is within the last FEP shared memory address slot. If the address range is legal, return *sharedBase* + *fepaddr*. Otherwise, return 0.

NOTE: Once the shared memory addresses and ranges are defined, this implementation may change.

Postconditions:

To produce sensible results, the client is responsible for ensuring that the FEP is powered on before reading or writing to the returned address.

Concurrency: Synchronous

10.7.8 powerOff()

Public member of: **FepDevice**

Return Class: **void**

Documentation:

Disable power to the Front End Processor, using *bepReg*.clrControl().

Postconditions:

Power is removed from the FEP and any information stored in FEP memory is lost.

Concurrency: Synchronous

10.7.9 powerOn()

Public member of: **FepDevice**

Return Class: **void**

Documentation:

This function enables power to the Front End Processor, using *bepReg*.setControl().

Postconditions:

Power is applied to the FEP. If the power was off prior to this call, the FEP will come up in a reset state, and any information stored in FEP memory prior to this call is lost. If power was already applied to the FEP prior to this call, asserting this bit has no effect.

Concurrency: Synchronous

10.7.10 releaseReset()

Public member of: **FepDevice**

Return Class: **void**

Documentation:

De-assert the Front End reset line, using **FepDevice**::clrControl(), allowing the FEP to boot and run.

Preconditions:

The FEP must be powered on, and the BEP must have initialized the FEP's microboot control word and stored executable code at the FEPs reset vector location.

Postconditions:

If the FEP's reset line was asserted prior to this call, it will configure its memory according to the microboot control word, and start executing from its reset vector. If the FEP's reset line was not asserted prior to this call (i.e. the FEP is already running), de-asserting this bit has no effect.

Concurrency: Synchronous

10.7.11 selectCcd()

Public member of: **`FepDevice`**

Return Class: **`void`**

Arguments:

`CcdId` *`ccdid`*

Documentation:

This function selects which CCD is processed by the associated Front End Processor. *`ccdid`* identifies which CCD's data is processed by this FEP.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Disable interrupts by declaring **`IntrGuard`** instance, *`guard`*. Use `clrControl`() to zero all CCD bits in the FEP's control register. Then shift *`ccdid`* to the appropriate bit position and pass to `setControl`(), effectively storing *`ccdid`* without modifying other bits in the control register. Once this function returns, *`guard`*'s destructor will restore the previous interrupt state.

Postconditions:

This FEP will accept image data only from the CCD specified by *`ccdid`*.

Concurrency: Synchronous

10.7.12 setControl()

Protected member of: **FepDevice**

Return Class: **void**

Arguments:

unsigned *mask*

Documentation:

This function sets the bits, designated by 1's in the *mask* argument, in the Front End Processor Control Register associated with this **FepDevice** instance. 0's in *mask* have no effect.

Preconditions:

For sensible results, the FEP must be powered on.

Semantics:

Declare **IntrGuard** instance, *guard*, to save interrupt state and disable interrupts. Read the BEP to FEP control register from shared memory (*sharedBase*[CTL_OFFSET]), bitwise-OR *mask* to clear the specified bits, and write the result back out to the control register. Once this function returns, *guard*'s destructor will be invoked, and will restore the previous interrupt state.

Concurrency: Synchronous

11.0 Detector Electronics Assembly Device (36-53211 A+)

11.1 Purpose

The purpose of the Detector Electronics Assembly (DEA) Device is to provide access to Back End's DEA interface logic and registers.

NOTE: The responsibility for forming command words, and interpreting reply words is lies with the ***`Protocols`*** layer, specifically, by classes operating under the direction of the **`DeaManager`**. See Section 26.0.

11.2 Uses

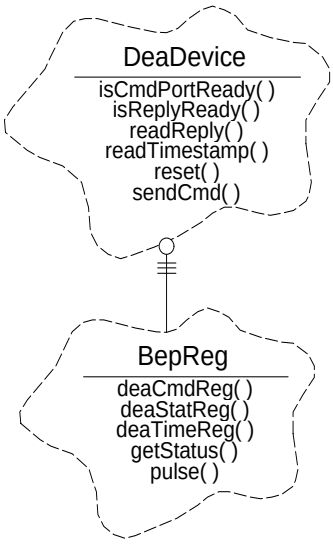
The DEA Device class provides the following features:

Use 1:: Send a command to the DEA
Use 2:: Read a response from the DEA
Use 3:: Read the timestamp latched by the previous command sent to the DEA

11.3 Organization

Figure 36 illustrates the relationships used by the **`DeaDevice`** class.

FIGURE 36. DEA Device Class Relationships



`DeaDevice`- This class is responsible for providing access to the Back End Processor's Detector Electronics Assembly command and reply port hardware, and the microsecond science timestamp. This class provides functions to reset the DEA interface card, effectively turnning the power off to the remaining DEA CCD controller boards (`reset`()), test the status of the command and reply ports (`isCmdPortReady`(),

`isReplyPortReady`()), write commands to and read replies from the DEA (`sendCmd`(), `readReply`()), and read the microsecond timestamp value, latched when a command is sent to the DEA (`readTimestamp`()). This class uses the **`BepReg`** class to get access to the DEA interface register addresses (`deaCmdReg`(), `deaStatReg`(), `deaTimeReg`()), to read the command port and reply port status (`getStatus`()), and to reset the reply port ready bit (`pulse`()).

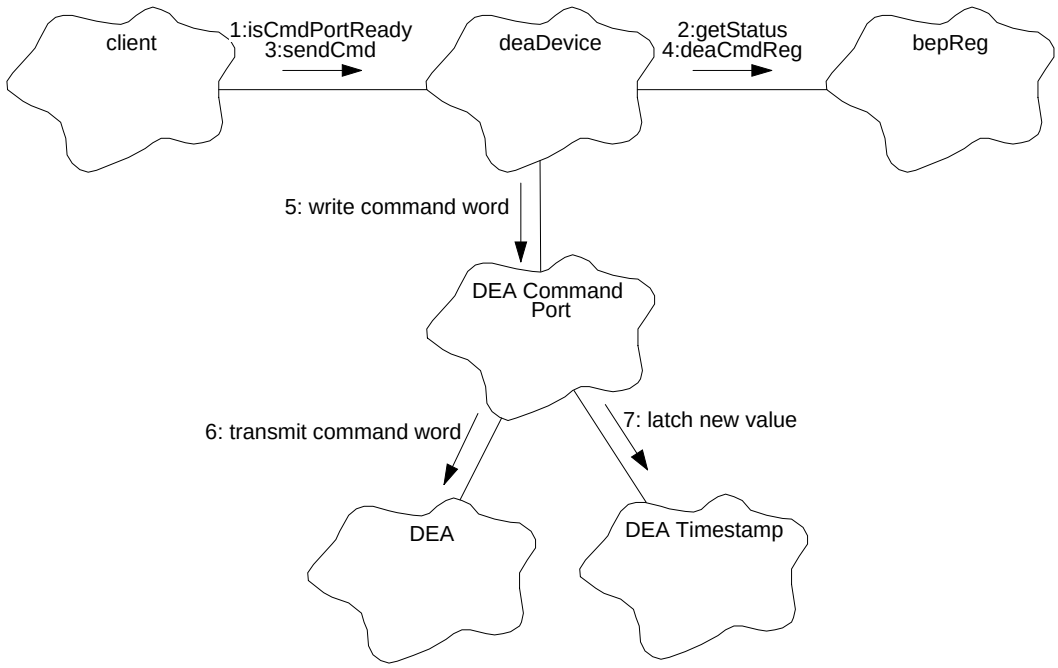
`BepReg`- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers. For more detail, see Section 5.0.

11.4 Scenarios

11.4.1 Use 1: Send a command to the DEA

Figure 37 illustrates the steps used to send a command to the DEA.

FIGURE 37. Send command to DEA



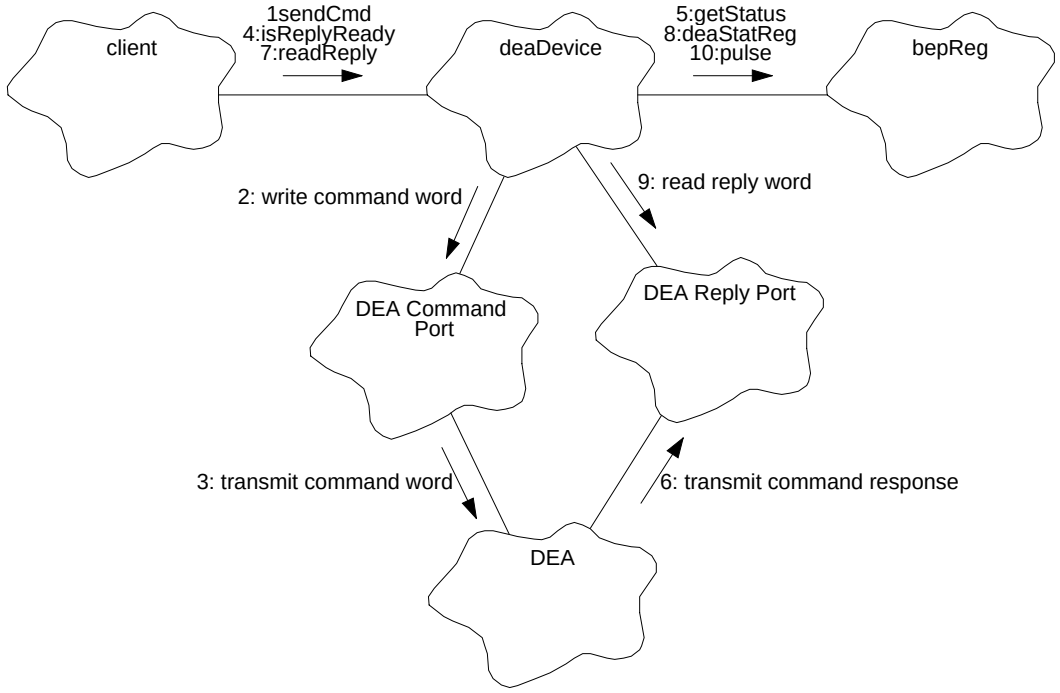
1. *`client`* polls the DEA Device until the command port becomes available, using *`deaDevice`*.`isCmdPortReady`(). NOTE: It is *`client`*'s responsibility to detect and handle time-outs on the DEA interface.
2. `isCmdPortReady`() uses *`bepReg`*.`getStatus`() to read the BEP's Status Register. It tests the DEA Command Port Available bit in the returned status word, and returns whether or not the port is busy.
3. Once the port is ready, *`client`* tells the DEA Device to send a command word to the DEA, using *`deaDevice`*.`sendCmd`().

4. `sendCmd`() uses *`bepReg`*.`deaCmdReg`() to obtain the address of the DEA Command Port.
5. `sendCmd`() then writes the command word into the port.
6. The command port hardware then proceeds to serially clock the word out to the DEA.
7. Once the command word has been clocked to the DEA, the interface hardware latches the current microsecond counter into the DEA Timestamp register.

11.4.2 Use 2: Read a response from the DEA

Figure 38 illustrates the steps used to read a reply word, sent by the DEA in response to a command.

FIGURE 38. Read command reply



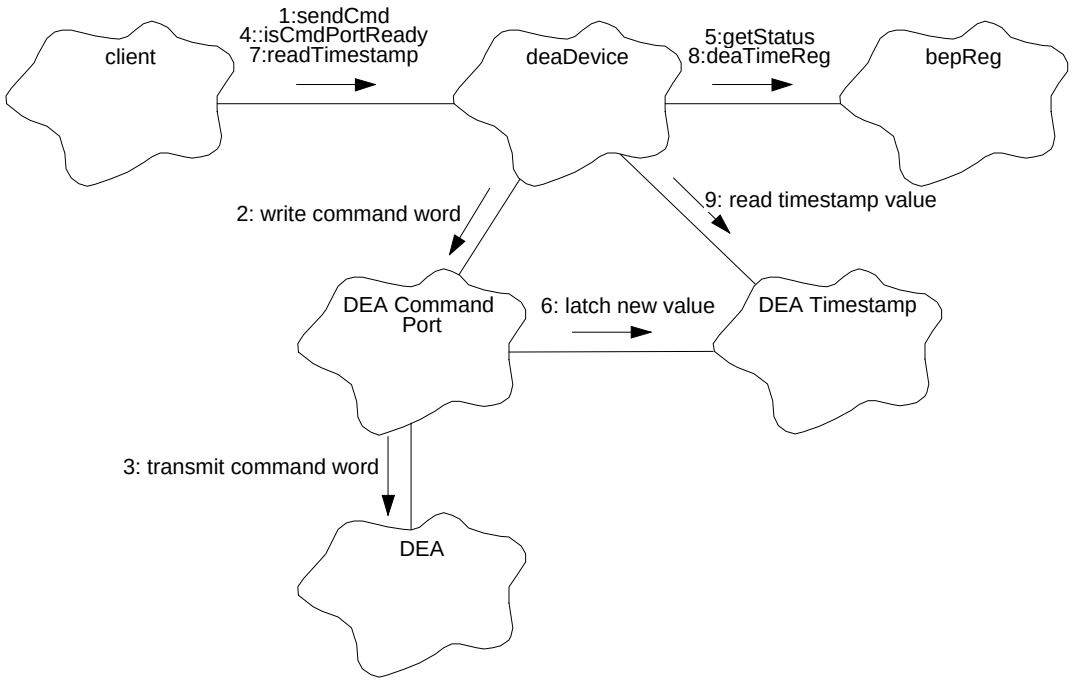
1. *`client`* sends a command to the DEA (which is intended to illicit a response) using *`deaDevice`*.`sendCmd`() (see Section 11.4.1for more detail on this action).
2. `sendCmd`() writes the command word to the DEA Command Port.
3. The DEA command hardware transmits the command word to the DEA
4. Meanwhile the *`client`* polls the reply status using *`deaDevice`*.`isReplyReady`(). NOTE: It is *`client`*'s responsibility to detect and handle time-outs on the DEA interface.
5. `isReplyReady`() uses *`bepReg`*.`getStatus`() to read the BEP's Status Register. It then tests the DEA Reply Ready bit, and returns whether or not a response is in the DEA's Reply Port.

6. Eventually, the commanded DEA board generates a response to the sent command, and sends it back to the BEP's DEA Reply Port.
7. Once the reply is received, `isReplyReady`() indicates that a response is available. *`client`* then reads the reply word using *`deaDevice`*.`readReply`().
8. `readReply`() uses *`bepReg`*.`deaStatReg`() to get the address of the DEA Reply Port.
9. `readReply`() reads the value from the reply port.
10. `readReply`() then resets the Reply Ready bit in the BEP's Status Register, using *`bepReg`*.`pulse`(). It then returns the read reply word to *`client`*.

11.4.3 Use 3: Read the latched timestamp

Figure 39 illustrates the steps used to read the latched DEA command timestamp.

FIGURE 39. Read DEA Command Timestamp



1. *`client`* sends a command to the DEA using *`deaDevice`*.`sendCmd`().
2. `sendCmd`() writes the command word to the DEA Command Port
3. The command port hardware transmits the command word to the DEA
4. Meanwhile, *`client`* polls the command port status, using *`deaDevice`*.`isCmdPortReady`(). Once the port becomes available, the command word will have been completely sent, and the timestamp value will be valid.

5. `isCmdPortReady`() uses *`bepReg`*.`getStatus`() to obtain the command port status.
6. Eventually, the command word transmission will be complete, and the DEA interface hardware will latch the current microsecond counter into the DEA Timestamp register.
7. Once `isCmdPortReady`() indicates that the command has been completely sent, *`client`* calls *`deaDevice`*.`readTimestamp`() to read the latched microsecond timestamp value. NOTE: It is the caller's responsibility to detect and handle time-outs on the DEA interface.
8. `readTimestamp`() uses *`bepReg`*.`deaTimeReg`() to get the address of the timestamp register.
9. `readTimestamp`() then reads the contents of the timestamp register, and returns the read value to *`client`*.

11.5 Class DeaDevice

Documentation:

This class is responsible for sending commands to the Detector Electronics Assembly (DEA) and providing status information send by the DEA.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Implementation Uses:

BepReg

Public Interface:

Operations:
```
isCmdPortReady()
isReplyReady()
readReply()
readTimestamp()
reset()
sendCmd()
```

Concurrency: Guarded

Persistence: Persistent

11.5.1 isCmdPortReady()

Public member of: **DeaDevice**

Return Class: **Boolean**

Documentation:

This function tests the current status of the DEA command port (via **BepReg**::`getStatus`()). It returns *`BoolTrue`* if the DEA is able to accept commands and *`BoolFalse`* if the DEA is not ready to accept commands (i.e. the interface port is busy).

Concurrency: Synchronous

11.5.2 isReplyReady()

Public member of: **DeaDevice**

Return Class: **Boolean**

Documentation:

This function tests the BEP status register (accessed via **BepReg**::`getStatus`()) to determine the DEA status register contains a reply word supplied by one of the DEA boards. This function returns *`BoolTrue`* if there is a DEA reply word ready for reading, and *`BoolFalse`* if a reply has not yet been received.

Concurrency: Synchronous

11.5.3 readReply()

Public member of: **DeaDevice**

Return Class: **unsigned**

Documentation:

This function reads and returns the contents of the DEA Status Register (whose address is determined using **BepReg**::deaStatReg()), and clears the DEA Reply Ready bit using **BepReg**::pulse().

Preconditions:

The caller must ensure that a command which requests a reply has been sent at least TBD microseconds prior to invoking this function. See sendCmd() and isReplyReady(). This function DOES NOT CHECK to ensure that a reply is in the status register. If no reply is present, garbage may be returned to the caller.

Postconditions:

The DEA reply ready status will remain de-asserted until another command requesting information is sent.

Concurrency: Guarded

11.5.4 readTimestamp()

Public member of: **DeaDevice**

Return Class: **unsigned**

Documentation:

This function reads and returns the contents of the microsecond timestamp, latched when the last command was sent to the DEA. This function uses **BepReg**::deaTimeReg() to get the address of the timestamp register.

Preconditions:

The client must ensure that the timestamp is read after the command has been sent to the DEA, using isCmdPortReady().

Concurrency: Guarded

11.5.5 reset()

Public member of: **`DeaDevice`**

Return Class: **`void`**

Documentation:

This function pulses the master DEA reset control signal. This signal resets all values on the DEA controller board, which, in turn, shuts the power off to all of the CCD-controller boards. This function uses *`bepReg`*.`pulse`() to pulse the reset line.

Postconditions:

All registers in the DEA's interface board will be zero, and all CCD-controller boards will be off.

Concurrency: Guarded

11.5.6 sendCmd()

Public member of: **`DeaDevice`**

Return Class: **`void`**

Arguments:

`unsigned` *`cmdWord`*

Documentation:

This function writes *`cmdWord`* to the DEA's Command Interface port.

Preconditions:

The command port must be ready for use (see `isCmdPortReady`()). If not, a previous command may be corrupted by *`cmdWord`*.

Postconditions:

If the command requested status information, this information will be available after TBD microseconds (see `isReplyReady`()).

Concurrency: Guarded

12.0 Radiation Monitor Device (36-53237 A)

12.1 Purpose

The purpose of the Radiation Monitor Device is to provide access to the radiation monitor interface logic.

12.2 Uses

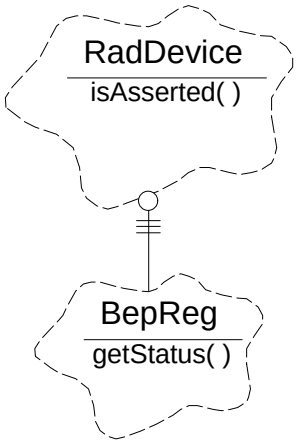
The Radiation Monitor Device class provides the following features:

Use 1:: Provide current state of the radiation monitor flag

12.3 Organization

Figure 40 illustrates the relationships used by the **`RadDevice`** class.

FIGURE 40. Radiation Monitor Device Class Relationships



`RadDevice`- This class is responsible for providing access to the Back End Processor's Radiation Monitor logic. This class provides a single function to test the current status of the monitor (`isAsserted`).

`BepReg`- This class represents the lowest level hardware access to the features provided by the Back End hardware control, status, and pulse registers. The **`RadDevice`** class uses this class to read the Back End's status register (`getStatus`).

12.4 Scenarios

12.4.1 Use 1: Provide current state of the radiation monitor flag

To obtain the current state of the radiation monitor flag, the client calls *radDevice*.isAsserted(). The member function isAsserted() then reads the Back End's status register, using *bepReg*.getStatus(). It then tests the radiation monitor status bit, and returns the appropriate answer to its caller.

12.5 Class RadDevice

Documentation:

This class represents the radiation monitor device. It provides access to the flag and handles radiation monitor interrupts.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: `None`

Implementation Uses:

`BepReg`

Public Interface:

Operations: `isAsserted()`

Concurrency: Synchronous

Persistence: Transient

12.5.1 isAsserted()

Public member of: **`RadDevice`**

Return Class: **`Boolean`**

Documentation:

This function indicates whether or not the radiation monitor is currently asserted by reading the Back End's status register using *`bepReg`*.`getStatus`(), and then testing the radiation status bit. If so, it returns *`BoolTrue`*. If the flag is not asserted, it returns *`BoolFalse`*.

Concurrency: Synchronous

13.0 Boot BEP (36-53231 A)

13.1 Purpose

The boot code provides the BEP with the capability to 1) boot from the Flight Software installed in memory; 2) to install code delivered from uplink and to begin executing at a selected location.

13.2 Uses

The Boot BEP provides the following feature:

Use 1:: Provides a means to initiate executable code on the BEP from the Flight Software.
Use 2:: To load and execute code from command packets.

13.3 Organization

13.3.1 Command Types

The booting of the BEP is normally controlled by external commands, either a power-on reset, or a discrete commanded reset. The latter command consists of two types, boot from memory and boot from uplinked command. An abnormal reset, triggered by the watchdog resetting, is expected when there has been an anomalous response from the running software. [1]

During boot, there is no capability for the software to echo commands to the ground. The setting of the BiLevel Discrete Downlink bits (LEDs on non flight boards) is used to indicate progress during the boot process. Their state is periodically sampled by the hardware at predetermined intervals. [2] The software “state” will continue to be reported by the main software after boot has transferred control. Refer to Section 13.6 Discrete Telemetry Status.

The standard mode of operation will use load from boot software in memory. The load from uplinked command is provided to allow ground to take over the BEP for maintenance, for debugging problems, for creating a functionality to perform a task not envisioned when the instrument was designed.

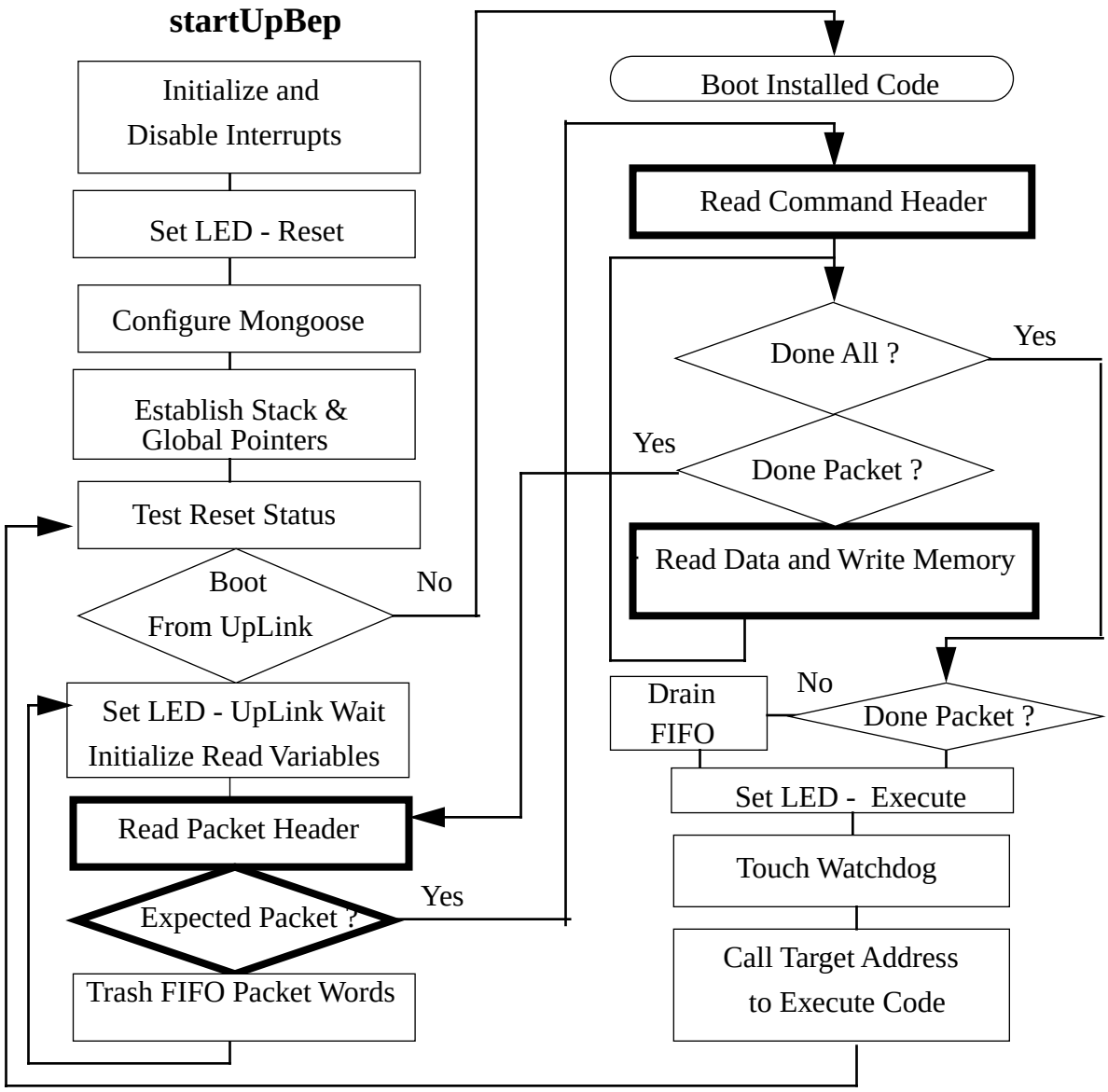
13.3.2 Boot Procedure

The boot procedure first manages the hardware interface including setting the BiLevel Discrete Downlink bits (LEDs) to indicate a reset, it then determines if a load from uplink is planned. If it is, the ground will have notified the instrument via the hardware command port, and the hardware will have set the BEP status register boot mode bit. If not loading from uplink, a standard boot

1. The hardware circuitry is described in *DPA Hardware Specification & System Description* Rev. B section 2.1.2.2
2. The Bilevel telemetry is described in *DPA Hardware Specification & System Description* Rev. B section 2.1.2.6.2

from memory will occur. Figure 41 illustrates the flow of control of the BEP boot procedure. The bolded boxes indicate functionality expanded in later figures.

FIGURE 41. BEP Boot Function Flow of Control



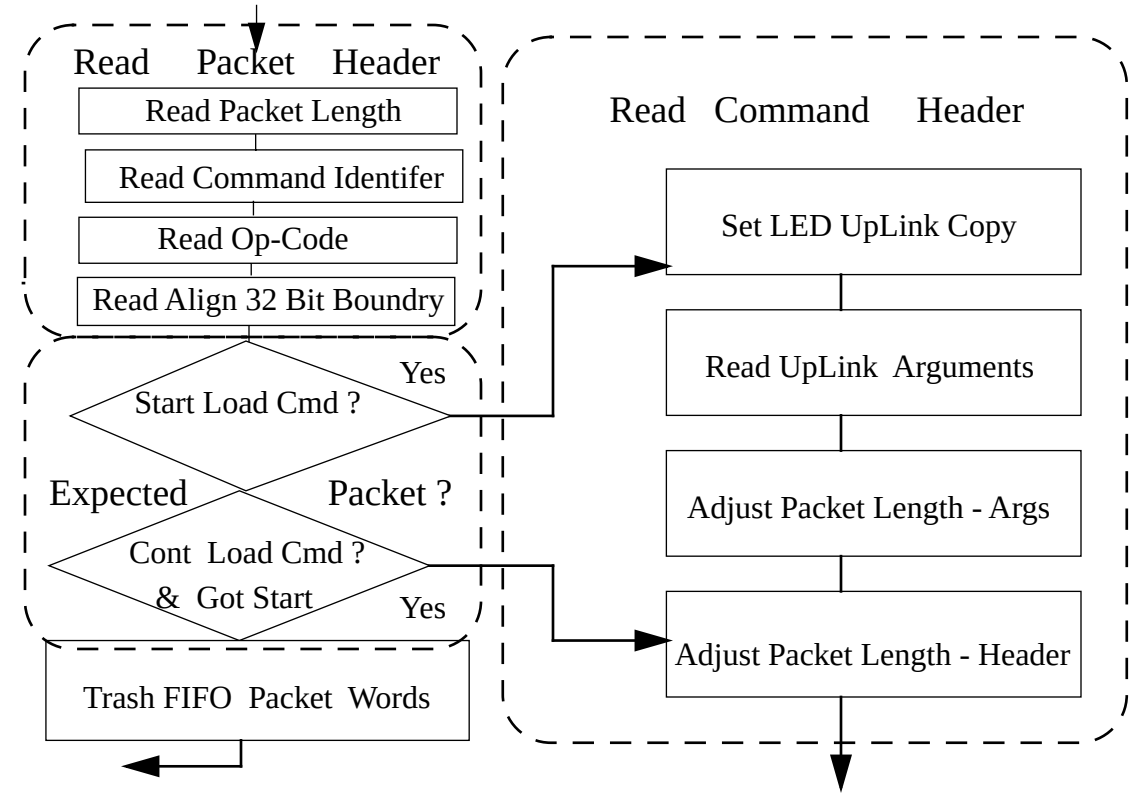
13.4 Scenario - Boot from Uplink Command

When booting from uplink the Uplink Wait LEDs are set, then the procedure will begin a loop which interrogates the FIFO as the commands arrive. During this cycle, the watchdog is handled. As the first command arrives the LEDs will be reset to Uplink Copy, indicating that data is being processed. The function will examine the packet header operation code and extract processing arguments from the command header. Unanticipated command packets will be discarded. The function will determine whether it should load data to memory or should execute code at an assigned address. It will load data to memory, cycling to obtain additional packets of data as nec-

essary. Should the data load request be fulfilled before the last packet is empty, the remainder of that packet's data will be drained from the FIFO. The LEDS will be reset to Uplink Execute showing that execution is beginning. The watchdog will be touched, giving the new process the maximum time before it must handle that task. Should the executing process return, the boot process will branch back to reinitialize the mongoose, and re-examine the BEP Boot Mode state.

The process will read the FIFO until it obtains the first packet word, which will contain the packet total length. It will read and ignore the command identifier, then read the op-code and determine whether it is a Continue_Load or a Start_Load command. Other commands are read from the FIFO and ignored. The process will read the align word from the FIFO. Refer to Figure 42.

FIGURE 42. Read Packet Header With Expected Packet Decision and Read Command Header

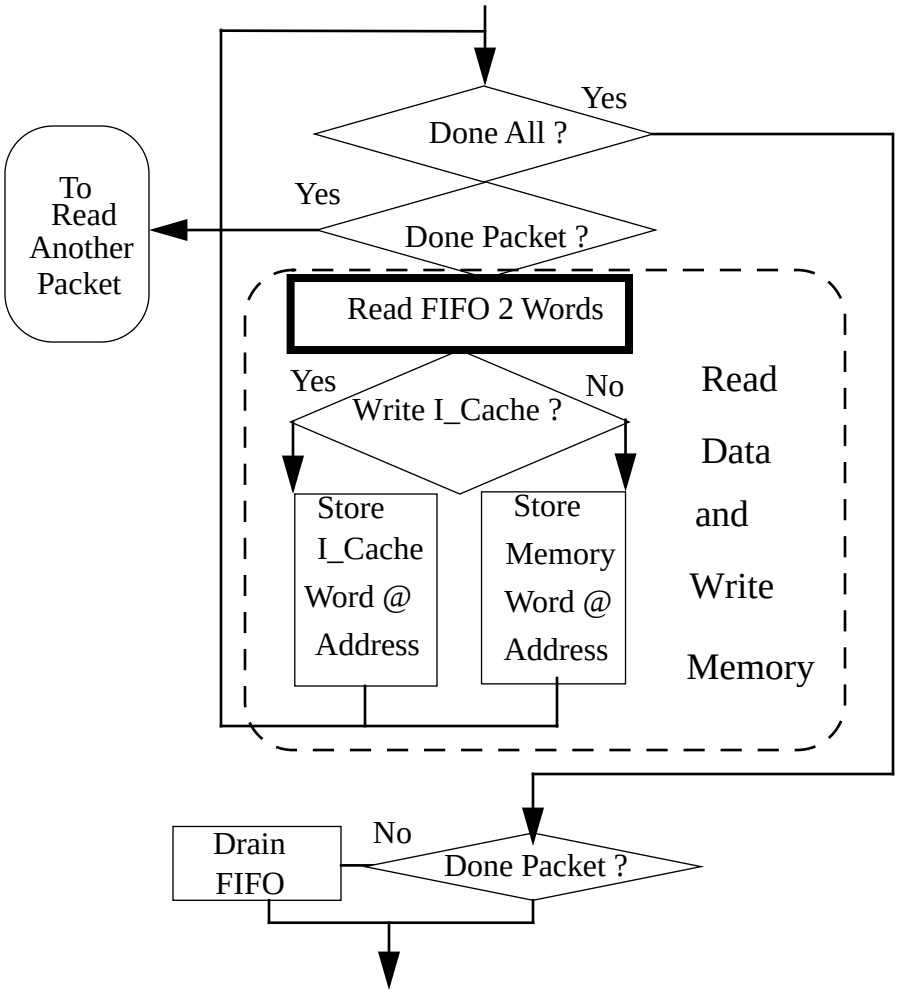


A Start_Load command will cause the LEDs to be set to indicate that an Uplink Copy is in progress. It will then read the total count of words to be written into memory and the address where execution is to begin after the data has been loaded. The packet length will be adjusted prior to advancing to a loop which reads the data and writes it to memory.

A Continue Load packet must be preceded by a Start_Load packet. When a Continue_Load command is recognized, the packet length is adjusted for the packet header size, before advancing to read the data and write it to memory. An unexpected packet op-code will result in trashing of that packet and restoration of the Uplink Wait (LED) state which requires a new Start_Load command before a Continue_Load command will be written to memory. An interviening Start_Load command takes precedence, and will load data as directed. This stratagem permits loading data to several locations prior to execution at its target address.

After having read the header, the process proceeds to the decisions concerning processing the remaining data. The total amount of data may have been loaded (including zero words). The amount of data in the packet may have been loaded, else the process will look for another packet. With more data to be loaded, the process will obtain the next word to be installed in memory. Refer to Figure 43. The bolded box indicates functionality expanded in the later figures.

FIGURE 43. Read Packet Data and Write It to Memory

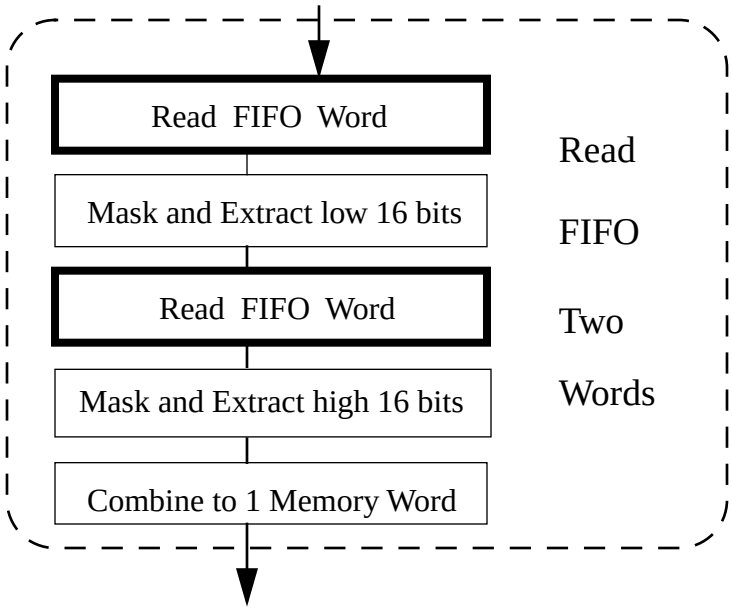


The I-Cache can not be written to directly. D-Cache and general memory can be written directly. To load to I-Cache the data to write, and the address to be written, must be delivered to hardware registers and the write will be accomplished. For this reason, the load address is tested and the data is then loaded in the appropriate manner. The process will return to evaluate the need to read and deposit another word.

After the Total number of words have been loaded, the process will check that the current packet has been emptied, else it will drain the FIFO of the residual words before advancing to begin execution.

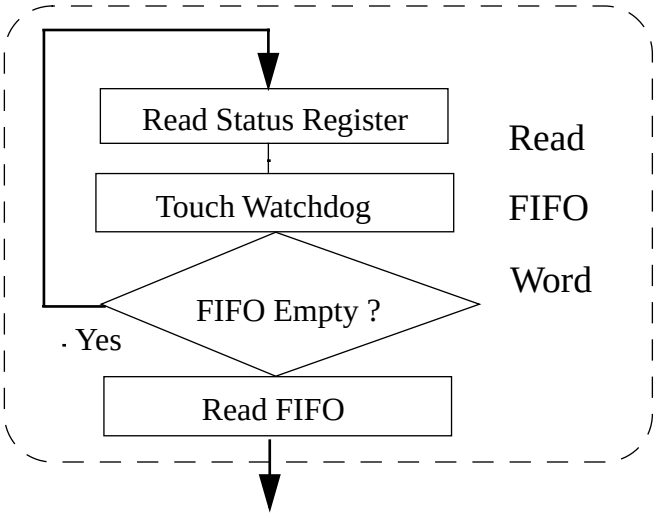
Each FIFO word contains 16 bits of data in the low two bytes, in addition to flags in the upper portion. When reading data to load into memory, two packet (FIFO) words are combined to create a single word to load into memory. Refer to Figure 44. The bolded box indicates functionality expanded in a later figure.

FIGURE 44. Read a Memory Word



The FIFO may or may not have an available word to be read every time one is requested. For this reason the process will read the FIFO status register. Then it will touch the watchdog to prevent a reset, and will test the Uplink-FIFO-Empty status bit to determine if a word may be transferred. It will continue to cycle until a word is available. It will read the word, then return to the calling function. Refer to Figure 45.

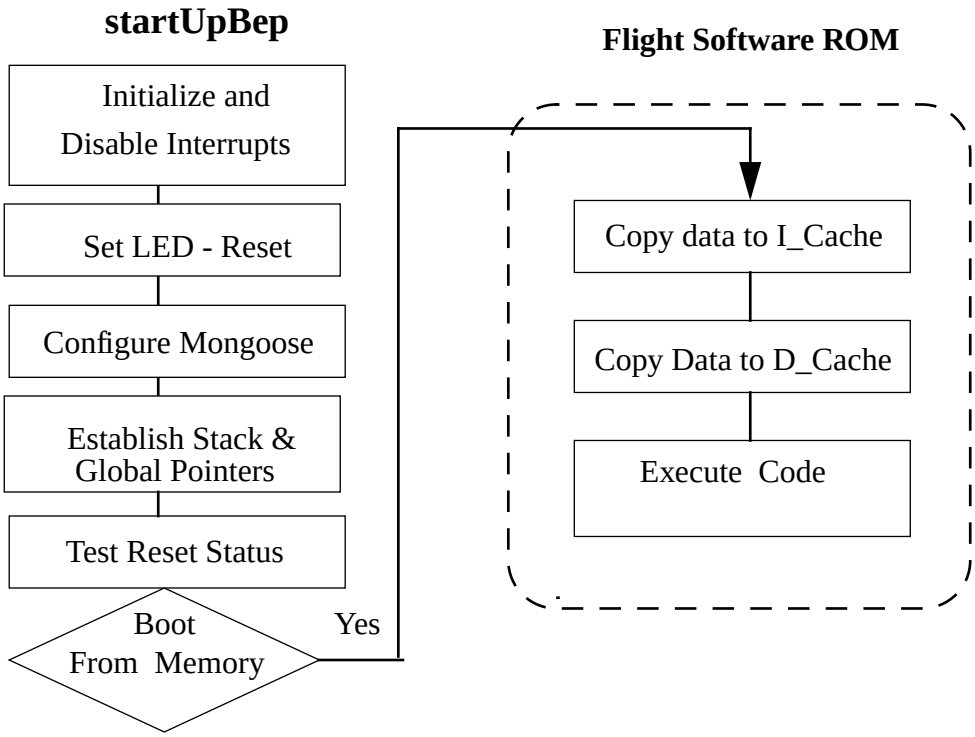
FIGURE 45. Read a FIFO Word



13.5 Scenario Boot from Flight Software in Memory

After initializing the mongoose, the process tests the reset status. If the mode indicates boot from FS in memory, control jumps to that memory location. Code stored in memory copies information into I-Cache (code) and D-Cache (data). It will then jump to the execution address. Execution from the FS memory is not expected to return. Refer to Figure 46.

FIGURE 46. Boot from Memory



13.6 Discrete Telemetry Status

The Back End's Boot from installed software and main application software use the four discrete telemetry bits to indicate the current status of the instrument software. These four bits allow the software to assign up to 16 different status values. These bits are sampled by the hardware once per Telemetry Major Frame.

The Boot software uses these bits to help a maintainer diagnose lockup problems during Boot. The Boot program asynchronously sets these bits to specific values at different stages during the boot. If the instrument hangs during the boot, the status bits will indicate which stage caused the hang. The stages and values are assigned as shown in Table 14.

TABLE 14. Boot Software Discrete Telemetry Value Assignments

Stage	Value	Description
Reset	15	The Back End Boot Code sets the status to this value upon reset of the Back End Processor. This value allows the maintainer to detect a lockup before the Boot process had a chance to detect the type of boot to be performed.
Memory Copy	14	The Back End Boot Code sets the status to this value prior to copying code and data from the Back End's Software. This value allows the maintainer to detect a failure while copying code and data from memory.
Memory Execute	13	The Back End Boot Software sets the status to this value prior to executing code loaded from the Back End's Software Memory. This value allows the maintainer to detect a lockup when the loaded code is executed.
Uplink Wait	12	The Back End Boot Code sets this value prior to polling the Command Interface for an "Start Load From Uplink" command. This value allows the maintainer to determine when the instrument is waiting for code and data from the uplink interface.
Uplink Copy	11	The Back End Boot Code sets this value after receiving the "Start Load From Uplink" command packet. This value allows the maintainer to determine that the instrument has received the first load command, and is in the process of copying its code and data, and waiting for subsequent "Continue Load From Uplink" commands.
Uplink Execute	10	The Back End Boot Software sets this value after receiving and loading all code and data from the Command Interface, and is about to execute the loaded code. This value allows the maintainer to detect a lockup when the loaded code is executed.

The loaded Back End software uses the status bits to help a maintainer determine the cur rent state of the instrument software. During its initialization stage, the Back End software sets the status bits to specific values to help a maintainer diagnose a hang during the initialization stage. After initialization, the Back End software periodically toggles the status between two values, depending on the current "state" of the software. Table 15 lists the values used by the Back End software.

TABLE 15. Main Software Discrete Telemetry Value Assignments

Stage	Value(s)	Description
Patch Application	9	The loaded Back End software sets the status bits to this value prior to applying any software patches. This allows the maintainer to determine if the Back End hangs while installing patches.
Startup	8	The loaded and patched Back End software sets the status bits to this value prior to executing its module initialization routines. This allows the maintainer to determine if the Back End hangs during its main initialization process.
Idle	7,6	The Back End software periodically toggles between these two values, while running when a Science Run is not being performed. This allows a maintainer to determine that the instrument is active, but not performing any science operations.
Science	5,4	The Back End software periodically toggles the status bits between these two values, rate, when executing a Science Run. This allows a maintainer to determine that the instrument is executing a Science Run.
WD Idle	3,2	If the Back End recovers after a Watchdog Reset, the software uses these values when it is in an “Idle” state. This allows a maintainer to determine that a Watchdog Reset occurred.
WD Science	1,0	If the Back End recovers after a Watchdog Reset, the software uses these values when it is performing a “Science Run.” This allows a maintainer to determine that a Watchdog Reset occurred.

14.0 System Startup and Patch Management (36-53242 01)

14.1 Purpose

The purpose of the System Startup and Patch Management units are to initialize and patch the instrument software running on the Back End Processor.

14.2 Uses

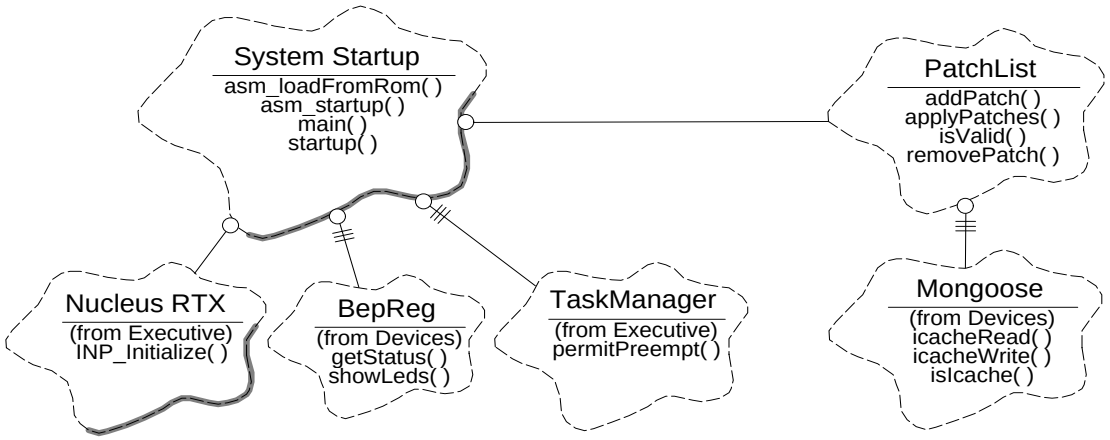
The following lists the use of the System Startup and Patch Management modules and classes:

Use 1:: Copy code and data from the bulk ROM into I-cache and D-cache on the BEP
Use 2:: Initialize processor registers and startup stack
Use 3:: Copy interrupt vector code into I-cache
Use 4:: Apply installed patches
Use 5:: Declare all global objects
Use 6:: Initialize and launch the real-time executive

14.3 Organization

Figure 47 illustrates the class relationships used by the **`System Startup`** routines, and the **`PatchList`** class. Although the **`System Startup`** routines are implemented in assembler or as stand-alone functions, they are represented as member functions of a class-utility in the figure (as indicated by the shadow on the cloud).

FIGURE 47. System Startup and PatchList relationships



`System Startup` - This is a collection of assembler and C++ subroutines which load and initialize the instrument software. They provide the ability to load code and initialized data sections from the bulk ROM into the Back End Processor's Instruction and Data cache (`asm_loadFromRom`), perform low-level processor initialization (`asm_startup`), perform patch installation, executive initialization and launch the executive (`startup`), and invoke global constructors once the executive is running and enable execution of the remaining tasks in the system (`main`). This class uses the **`BepReg`** class to determine if the most recent reset was a commanded reset, and if so uses the **`PatchList`** class to install the system patches.

`PatchList` - This class is responsible for maintaining the system patch list, and for installing the patches when invoked by startup. It provides functions which append a patch to the system patch list (`addPatch`), remove a patch from the list and compact the list (`removePatch`), determine if the patch list has been corrupted (`isValid`), and apply the set of patches during startup (`applyPatches`).

`Nucleus RTX` - This represents the collection of routines provided by the Nucleus Real-Time Executive. The startup software uses `INP_Initialize`() to initialize and launch the executive.

14.4 Startup and Patch List Design Issues

14.4.1 Patch List Memory Map and Organization

The system patch list is maintained in the Instruction Cache RAM (I-cache) on the Back End Processor (BEP). Since writing to I-cache must be accomplished via the Mongoose Command Status Interface, corruption of the list due to stray pointers or crashes is unlikely. However, whenever the BEP receives a power-on reset, or if the BEP is reset while removing a patch from the patchlist, there is the potential for the list to become corrupted. As such, the list maintains a checksum of the contents of the list. If the checksum is invalid, patches will not be applied.

Table 16 illustrates the layout of the system patch list. This list grows backwards, from the end of I-cache toward the start of I-cache.

TABLE 16. I-cache Patch List Layout

Region		Address	Byte Size	Description		
Patch Area	End of List Pointer	0x800ffffc	4	This points to the next location where a patch may be added		
	Checksum	0x800ffff8	4	This is the current 32-bit XOR checksum of the current contents of the patch list.		
	Patch Nodes		Variable	These are a collection of patch nodes. The format for each node is as follows:		
				Name	**Offset**	**Bytes**	**Description**
				Patch Id	4*Length +8	4	Unique code to identify the patch
				Destination	4*Length +4	4	Points to where to write patch data
				Length	4*Length	4	Number of 32-bit words in patch
				Data	0	4*Length	Data to write at startup
	Current End of Patch List	End of List Pointer		This is next location after the last patch node in the patch list. This location is pointed to by the End of List Pointer		
	...					
	Lowest Patch Address	0x800d7c00		This is the lowest address usable by the patch list. It cannot be moved without a patch.		
Bad Pixel/Column Maps		0x800cac00	53248			
Compression Tables		0x800c2c00	32760			
Parameter Blocks		0x800c0400	10240			
System Configuration		0x800c0000	1024			
Code		0x8008000	262144			

14.4.2 Bulk ROM Memory Map and Organization

The ACIS Bulk Flight ROM consists of a boot section and simple loading routine, which has been linked with the flight code, thus providing it with the locations to install the loadable code and initialized data sections. The code is assumed to precede data in a contiguous area pointed to by the symbol `_loadRom` (TBD). During system ROM boot, the Back End Processor's Boot Code jumps to the ROM loader. The loader ROM proceeds to copy code and data sections from the ROM into the BEP's I-cache and D-cache. The code is to be copied into I-cache from `_ftext` to `_etext`, while the data is to be located in D-cache from `_fdata` to `_edata`. Once the sections have been copied, the loader jumps to the starting execution address of the loaded code designated by `_execAddr` (TBD).

14.4.3 Executive Configuration

Nucleus RTX is configured using fixed tables at fixed locations. Table 17 lists the items which must be initialized prior to starting the executive. Refer to the Nucleus RTX include file, **in_defs.h**, for the definitions of these structures, and the definition of the END_OF_LIST constant.

TABLE 17. Nucleus RTX Configuration Items

Name	Type	Description
SKD_System_Stack_Ptr	**unsigned***	This must be initialize to point to the start of the system stack space used by the executive.
IN_System_Stack_Size	**unsigned**	This must be initialize to the total number of 32-bit words to use for the system stack.
IN_Last_Memory_Address	**unsigned**	This variable must be initialized to the last memory location to be used by the executive. The executive uses the area between the address of *IN_Last_Address_Used* and the location referenced by this variable for stacks, task control structures, etc.
IN_Last_Address_Used	**unsigned**	This variable must be initialized to reference the first location used by Nucleus RTX for stacks, task control structures, etc.
IN_Fixed_Partitions	**struct IN_FIXED_PARTITION_STRUCT[]**	This is an array of partition definition structures. The first field of entry just after the last used entry must contain the value “END_OF_LIST.” This array must contain an initialized entry for each **MemoryPool** instance. The index of the entry serves as the RTX identifier for the pool.
IN_System_Queues	**struct IN_QUEUE_DEFINITION_STRUCT[]**	This is an array of queue definition structures. The first field of the entry just after the last defined queue must contain the value “END_OF_LIST.” This array must contain an initialized entry for each **Queue** instance in the system. The index of the entry serves as the RTX identifier for the queue.
IN_System_Event_Groups	**unsigned**	This is the count of event groups used by the instrument. For ACIS, this must be initialized to the total number of tasks in the system.
IN_System_Resources	**unsigned**	This is the count of semaphores used by the instrument. For ACIS, this must be initialized to the total number of **Semaphore** instances used by the system
IN_System_Tasks	**struct IN_TASK_DEFINITION_STRUCT[]**	This is an array of task definition structures. The first field of entry just after the last defined task must contain the value “END_OF_LIST.” There must be one entry for each defined **Task** instance within the instrument. The index of an entry serves as the RTX identifier for the task.

14.4.4 Global Constructors

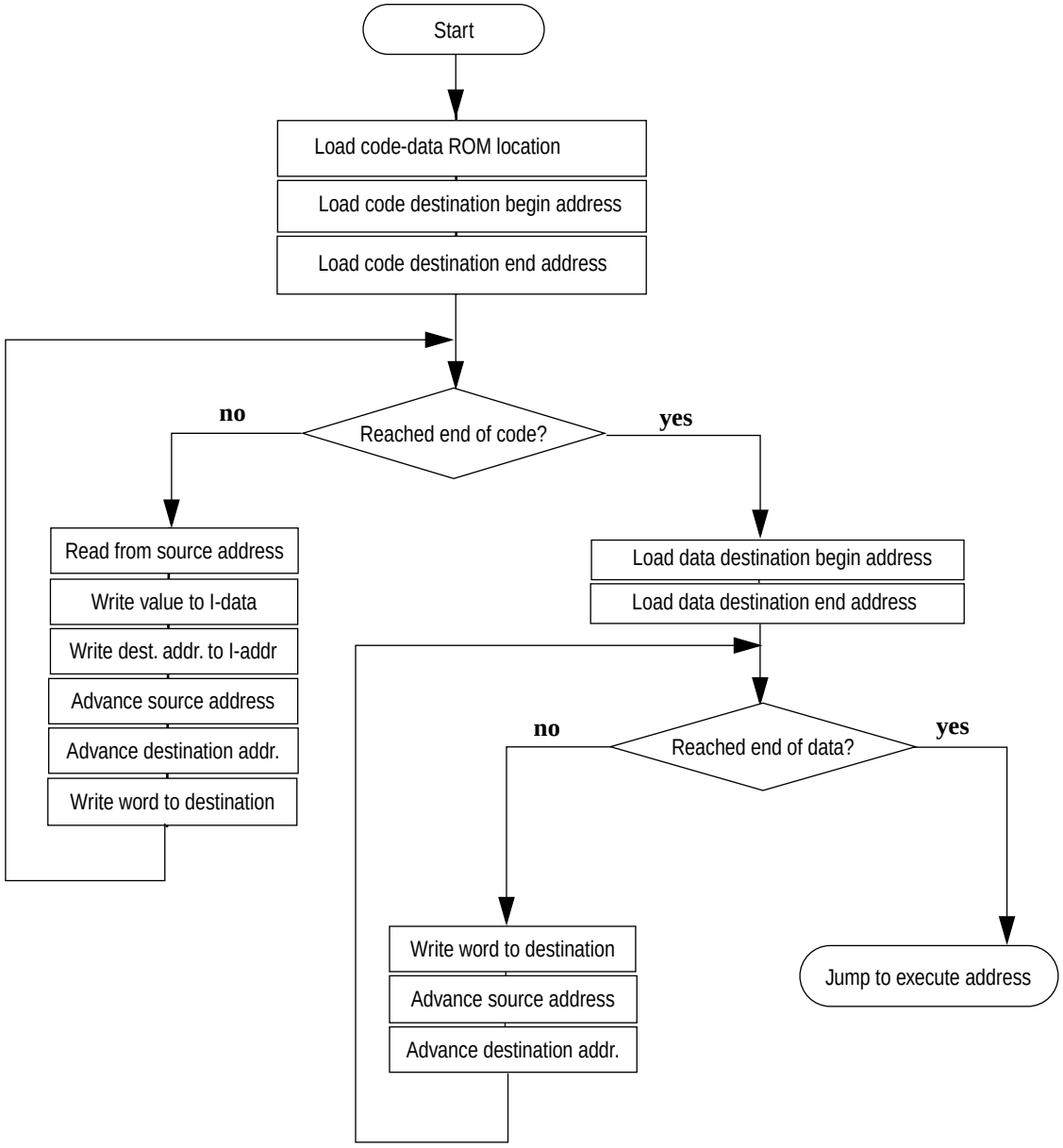
In C++, the constructors for global instances are invoked in the order they appear in an object file. The order in which different object files are processed is undefined by the language. In order to ensure deterministic initialization order, the **System Startup** uses a single source file for all global objects, **globals.C**. This file contains the Nucleus RTX initialization structures, and each global declaration, in the order they are to be initialized.

14.5 Scenarios

14.5.1 Use 1: Load code and data from the bulk ROM into the BEP

Figure 48 illustrates the sequence of operations which load code and data from the bulk ROM into the Back End Processor RAM. Note: the ROM locations of code and data and the areas in I-cache and in D-cache where they will be located, were compiled into the loader.

FIGURE 48. Loading Code and Data



14.5.2 Use 2: Initialize processor registers and startup stack

The code located at the startup execution address is responsible for initializing the some of the Mongoose and R3000's processor registers and setting up a stack for the C++ startup routines. Table 18 lists the register values which must be initialized. The R3000 symbols are defined in **mips.h**, and the Mongoose register symbols are defined in **mongoose.h**. (NOTE: By convention, C and C++ assume that uninitialized data section, .bss, be zeroed during startup, ACIS performs this action after the patch list has been applied. Code prior to and during patching shall not rely on uninitialized data having a value of zero).

TABLE 18. Startup Mongoose and R3000 Register Initialization

Register	Initial Value	Description
C0_SR (C0:$12)	SR_BEV | SR_CU0	R3000 Co-processor 0 Status Register: This register must be initialized to use the Boot Exception Vector, and to have Co-processor 0 enabled. During startup, all R3000 device interrupts are disabled.
C0_CAUSE (C0:$13)	zero	R3000 Co-processor 0 Cause Register: Writing a zero to the cause register ensures that there are no software interrupts pending.
M_CFGREG (Mongoose)	CR_NODMA | CR_WAITST1	Mongoose Configuration Register: This register is initialized to ensure that no Mongoose DMA transfers are underway, and that the processor use 1 wait-state for all bulk memory and devices.
M_MASK (Mongoose)	zero	Mongoose Interrupt Mask Register: Writing a zero to the Mongoose interrupt mask disables all Mongoose device interrupts.
gp (R3000:$28)	_gp	R3000 Global Pointer: This register is initialized to the linker defined value for the global pointer.
sp (R3000:$29)	startup_stack + stack size - 24	R3000 Stack Pointer: The stack pointer is initialized to point at the end of the stack, minus 24 bytes to support the minimum reserved stack space required by functions which call other functions (i.e. non-leaf functions).

Once the registers have been initialized, the startup assembler code jumps to the C++ startup routine, `startup`().

14.5.3 Use 3: Copy interrupt vector code into I-cache

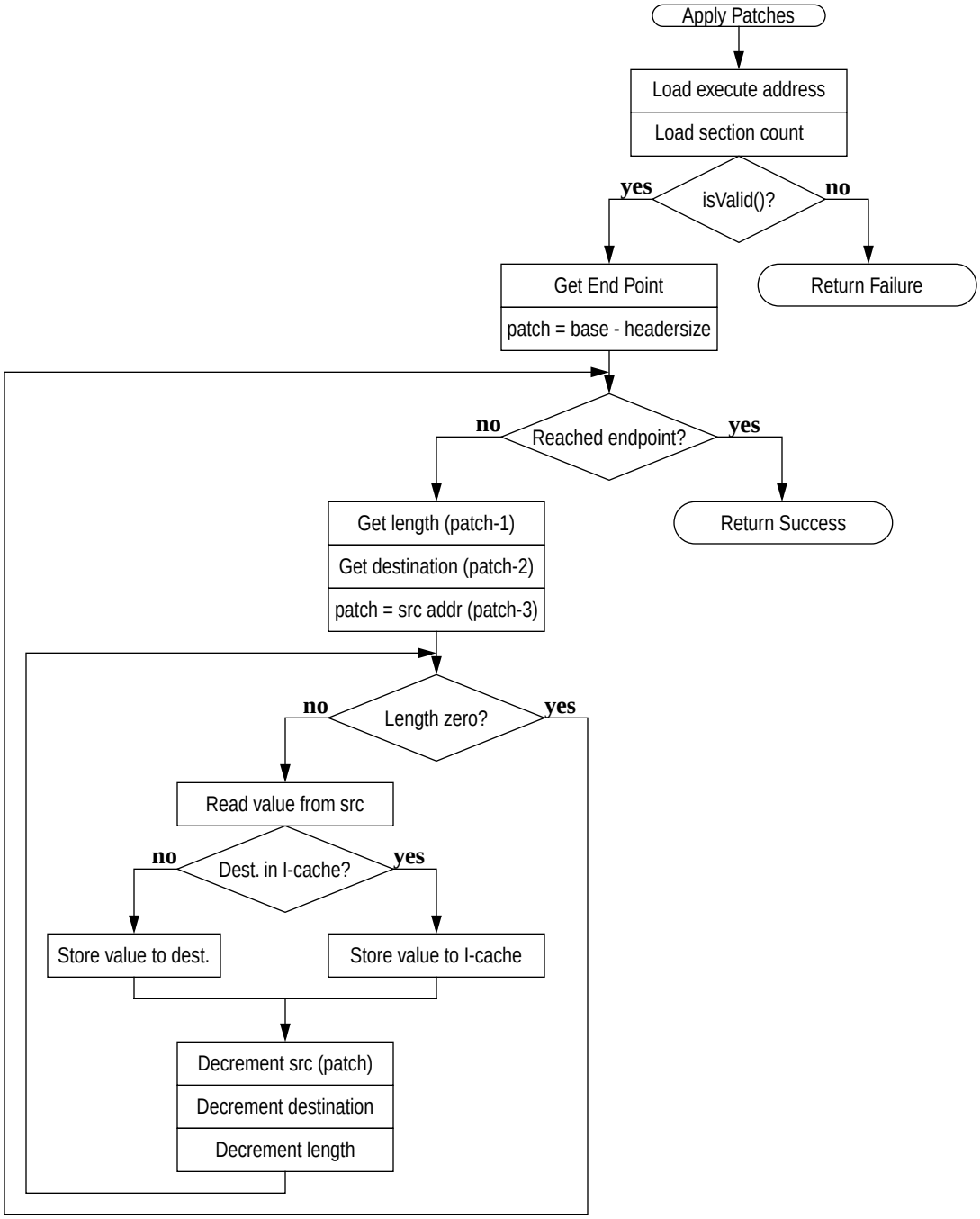
Prior to enabling interrupts on the R3000, the ACIS software must copy a code-fragment which jumps to the ACIS interrupt handler into the General Exception area of I-cache. On ACIS, this is accomplished by providing the appropriate code section in the bulk ROM (see Section 14.4.2 and Section 14.5.3). This causes the interrupt vector code to be loaded into I-cache when all of the other code and initialized data sections are loaded.

During the development process, however, when code is loaded via the ROM monitor across a serial interface, rather than from the instrument's bulk ROM, the startup code is required to copy the interrupt vector code-fragment into I-cache. This is because the ROM monitor uses non-boot interrupts when loading the images. The installed R3000 interrupt handler must then forward exceptions and UART interrupts to the original monitor's handler.

14.5.4 Use 4: Apply installed patches

The first action taken by the C++ startup routine is to determine if the instrument was commanded to reset, and if so, install the patch list. The startup routine uses *bepReg*.`getStatus`() to obtain the value of the Back End Processor's status register. If processor was commanded to reset (rather than watchdog reset, or power-on reset), startup calls *patchList*.`applyPatches`() to install the contents of the patch list. Figure 49 illustrates the process used to install the patch list.

FIGURE 49. Install patches



14.5.5 Use 5: Declare all global objects

In order to ensure a deterministic initialization sequence, the **`System Startup`** unit declares all global class instances in one file, **globals.C**. In general, system is initialized from the bottom up. The device classes are declared first, followed by the executive classes, followed by command handlers, and other protocols classes. Finally, the application classes, are declared. The exact order of the global class declarations (and subsequent initialization) is determined by the final implementation, and shall be provided in the AS-BUILT Detailed Design (MIT 36-53200).

14.5.6 Use 6: Initialize and launch the real-time executive

The **`System Startup`** uses patchable, initialized Nucleus RTX Configuration structures to configure the system's tasks, semaphores, and event groups. Since the memory partitions and queues are coupled to and rely entirely on the allocation of telemetry buffers in the Back End Processor's bulk memory, the **`System Startup`** uses a telemetry buffer configuration table to determine which telemetry buffer pools are required, the size of the packets maintained by each pool, and the number of buffers in each pool. The main initialization routine, `startup`(), calls `setupRtx`() to initialize the RTX memory partition and queue structures corresponding to each telemetry buffer pool, and establish the total number of telemetry packets in the instrument. This total is then used to initialize the RTX queue used for the telemetry manager's (**`TlmManager`**) telemetry packet buffer transmission queue.

14.6 System Startup Routines

Documentation:

This is a collection of assembler and C++ subroutines which initialize the instrument software, apply the patchlist, and launch the real-time executive.

Export Control: Public

Uses:

PatchList
Nucleus RTX
BepReg
TaskManager

Interface:

Operations:

`asm_loadFromRom()`
`asm_startup()`
`main()`
`setupRtx()`
`startup()`

Concurrency: Sequential

Persistence: Transient

14.6.1 asm_loader()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is an R3000 assembler routine which runs directly from the bulk ROM. This routine copies the code and data sections from the bulk ROM into the Back End Processor's Instruction and Data caches.

Concurrency: Sequential

14.6.2 asm_startup()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is an R3000 assembler routine which initializes the R3000 processor registers, zeros the uninitialized data section of memory, .bss (beginning at the address indicated by the linker symbol “_bss” and ending at the address indicated by “_end”), sets up a stack for use during startup, installs the interrupt vector code at the start of I-cache, and branches to the main C++ startup code.

Concurrency: Sequential

14.6.3 main()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is the routine installed to be the first routine invoked when the executive starts. It is established as the system initialization thread by startup as a low-priority task, which executes upon starting the executive, and has preemption disabled. Once the executive starts, it invokes this task. The C++ compiler embeds a call to __main() within this function, which invokes all of the global constructor routines. Once the constructors have been invoked, this routine tells the *taskManager* to allow preemption. At this point the higher priority tasks are run. In case all tasks suspend, this function enters an infinite loop to prevent the task from returning to the executive.

Concurrency: Synchronous

14.6.4 setupRtx()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This function is responsible for setting up the Nucleus RTX configuration tables. The function iterates through the *tlmPoolConfig* table, declared in **globals.C**. For each entry in the table, it initializes the corresponding entry in the *IN_Fixed_Partitions* and *IN_System_Queues* tables, also declared in **globals.C**. It then adds the number of packets in the entry's pool to *totalPacketCount*. Once the entry table has been processed, it initializes remaining entry in *IN_System_Queues* for the telemetry manager's (*tlmManager*) packet transmission queue.

Concurrency: Sequential

14.6.5 startup()

Public member of: **`System Startup`**

Return Class: **`void`**

Documentation:

This is the main C++ startup routine. This function installs patches, initializes the real-time executive, and launches the executive. This routine uses *`bepReg`*.`getStatus`() to obtain the contents of the Back End Register's Status Register. If the status indicates a commanded reset, it calls *`patchList`*.`applyPatches`() to install the patch list. The function then calls `setupRtx`() to configure the telemetry packet buffer partitions and queues. Finally, `startup`() calls `INP_Initialize`() to initialize ***`Nucleus RTX`*** and start the executive. Once the executive is running, the only task enabled to run, main(), is invoked.

Concurrency: Sequential

14.7 Class PatchList

Documentation:

This class represents the list of ACIS patches.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:
```
addPatch()
applyPatches()
isValid()
removePatch()
```

Protected Interface:

Operations:
```
computeChecksum()
findPatch()
updateChecksum()
```

Private Interface:

Has-A Relationships:

`unsigned* const` *`patchBase`*: This is the base address of the patch area in I-cache.

Concurrency: Guarded

Persistence: Persistent

14.7.1 addPatch()

Public member of: **PatchList**

Return Class: **Boolean**

Arguments:

unsigned *patchId*
unsigned* *dstAddress*
unsigned *datalen*
const unsigned* *patchData*

Documentation:

This function adds a patch to the system patch list. *patchId* is the identifier for the patch to add. *dstAddress* is the destination address that the patch is written to when applied the next time the instrument receives a commanded reset. *datalen* is the number of 32-bit data words in the patch, and *patchData* points to the data belonging to the patch. If a patch already exists with the same *patchId*, the function returns *BoolFalse* and the patch is not installed. If no conflicting patch is found, the new patch is appended to the end of the patch list and the function returns *BoolTrue*.

Semantics:

This function concatenates the patch to the list, updates the checksum of the list and then advances the list endpoint. If a reset occurs before the checksum is stored and the endpoint is advanced, the patch is not installed. If a reset occurs after the checksum is stored, but before the endpoint is advanced, the entire list will be flagged as invalid.

Concurrency: Guarded

14.7.2 applyPatches()

Public member of: **PatchList**

Return Class: **Boolean**

Documentation:

This function checks the checksum of the patch list. If the list is valid, it traverses the list and applies each patch. If successful, the function returns *BoolTrue*. If the list is invalid, it returns *BoolFalse*. This function is implemented as a static member function to allow patches to be applied during startup before any of the C++ constructors are invoked. See Section 14.5.4 for a detailed description of the operation of this function.

Concurrency: Sequential

14.7.3 computeChecksum()

Protected member of: **PatchList**

Return Class: **unsigned**

Documentation:

This function computes the checksum of the current patchlist, and returns the computed value. This function uses mongoose.icacheRead() to obtain the current end of the patch list. It then sets the initial sum value to all 1s, and iterates through each word in the patchlist and XORs the word with the current sum. Once the list has been processed, the function returns the sum.

Concurrency: Guarded

14.7.4 findPatch()

Protected member of: **PatchList**

Return Class: **unsigned***

Arguments:

unsigned *patchId*

Documentation:

This function searches the patchlist for the patch entry associated with *patchId* and returns a pointer to the start of the patch. If no such patch is found, the function returns 0. The function uses *mongoose*.icacheRead() to obtain the end point of the list. It then traverses the list until it reaches the end point, or it finds a patch whose identifier matches *patchId*. If it finds a match, the function returns the address of the matching patch. If not, it returns 0.

Concurrency: Guarded

14.7.5 isValid()

Public member of: **PatchList**

Return Class: **Boolean**

Documentation:

This function checks the checksum of the patch list. If the list is valid, it returns *BoolTrue*. If the patch list has been corrupted, it returns *BoolFalse*. This function is implemented as a static member function to allow it to be used during startup, before any of the C++ constructors have been invoked. This function calls computeChecksum() to compute and return the checksum of the data currently stored, and compares the result with the value located in the patch list header. If the values match, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Concurrency: Guarded

14.7.6 removePatch()

Public member of: **PatchList**

Return Class: **Boolean**

Arguments:

unsigned *patchId*

Documentation:

This function searches the patch list for a patch containing patchId. If found, the function removes the patch from the list and returns *BoolTrue*. If the patch is not found, it returns *BoolFalse*.

Semantics:

Search until the node is found, then copy the remaining patch area on top of the removed patch. Update the checksum and adjust the end of list pointer. If a reset occurs during the compacting operation, the checksum will be invalid and the list will be flagged invalid.

Concurrency: Guarded

14.7.7 updateChecksum()

Protected member of: **PatchList**

Return Class: **void**

Documentation:

This function computes and stores the checksum of the current patch list. The new checksum is computed using computeChecksum(), and is stored using *mongoose*.icacheWrite().

Concurrency: Guarded

15.0 Real-Time Executive (36-53212 A)

15.1 Purpose

The Real-Time Executive (RTX) category provides a top level interface to the underlying real-time executive. This serves to isolate the rest of the application code from the details of the executive, and facilitates the development of additional executive features.

15.2 Uses

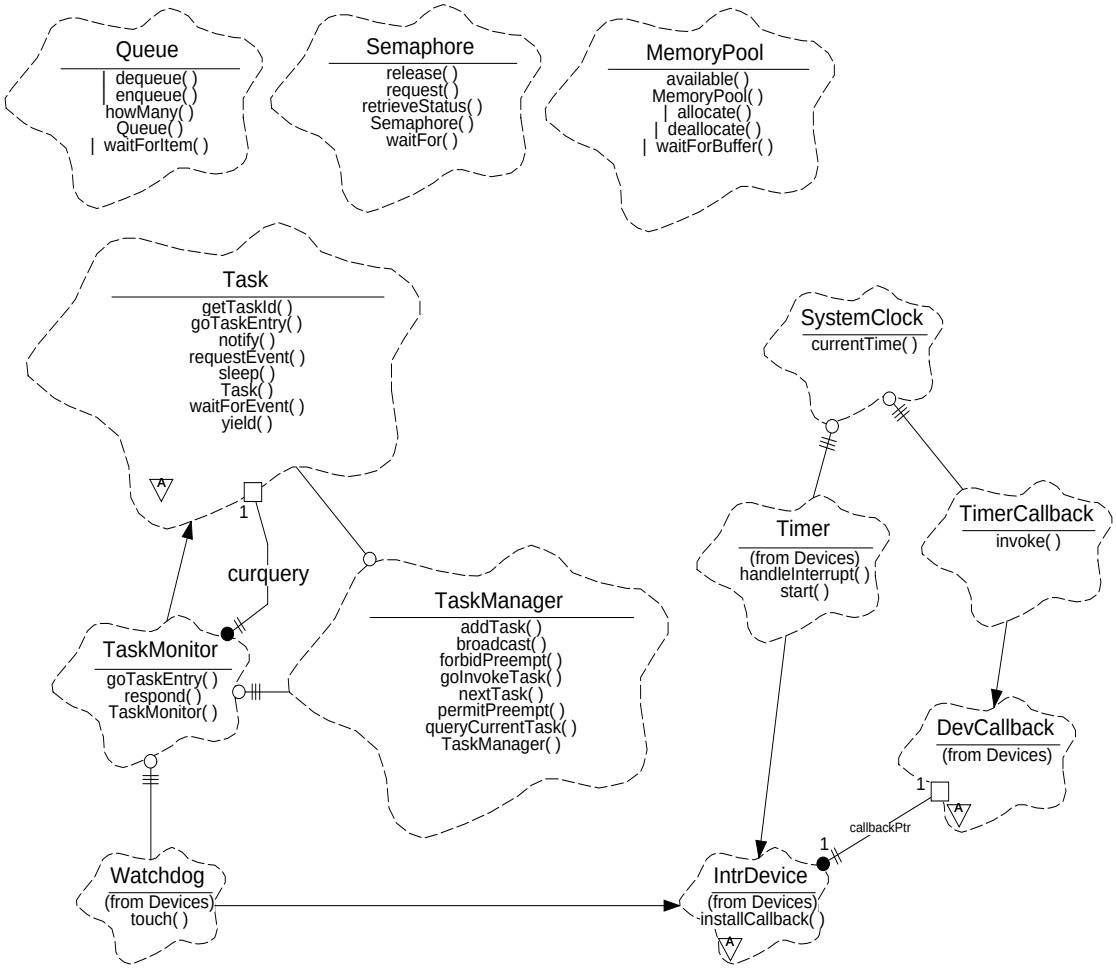
The Executive classes provide the following features:

Use 1:: Launch and maintain multiple preemptive tasks
Use 2:: Notify running tasks of discrete events
Use 3:: Broadcast events to all tasks within the system
Use 4:: Lock and unlock access to resources shared by more than one task
Use 5:: Maintain and use inter-process queues
Use 6:: Allocate and release buffers from fixed size buffer pools
Use 7:: Maintain and provide the number of timer ticks since the last reset
Use 8:: Perform “aliveness” tests on each running task

15.3 Organization

Figure 50 illustrates the top-level relationships between these classes provided by the executive interface.

FIGURE 50. Real-Time Executive Interface Class Diagram



The Real-time Executive system uses ***Nucleus RTX*** to implement most of its features. The executive also relies heavily on the ACIS start-up code to initialize the RTX structures, and launch the executive. ***Nucleus RTX*** itself relies heavily on the low-level interrupt handler and on a timer-tick interrupt (see Section 15.4).

Task - This abstract class represents a single, independent thread of control. Its implementation uses the underlying multi-tasking facilities of the RTX. This class also uses the event notification facilities of RTX to provide task suspension and notification features. This class provides functions to obtain its RTX task id (`getTaskId`), notify the **Task** of an event (`notify`), check to see if an event has been reported (`requestEvent`), suspend execution for a period of time (`sleep`), suspend execution until an event has been reported (`waitForEvent`), and suspend execution until all other tasks of the same priority have had a chance to run (`yield`). Each subclass of **Task** must implement a `goTaskEntry` function, which will be invoked by the *taskManager* during system start-up. Once the system is running, the only direct action performed on one task by another task through event notification (`notify`, `broadcast`).

`TaskManager` - This class is responsible for coordinating the activities of individual tasks. It uses the **`Task`** class to coordinate these activities. Within ACIS, there is only one global instance of the **`TaskManager`** class, called *`taskManager`*. This class provides a function used during start-up to add a new task (`addTask`), and a static member function invoked by Nucleus to start a particular task (`goInvokeTask`). This class provides run-time functions to broadcast events to all tasks (`broadcast`), to disable and enable task preemption in the currently active task (`forbidPreempt`, `permitPreempt`), iterate through each configured task (`nextTask`), and obtain a pointer to the currently running task (`queryCurrentTask`).

`Queue` - This class represents an a abstract queue of items. It's implementation uses the inter-process queue facilities of the RTX. This class provides functions used by its sub-classes to add items to the end of the queue (`enqueue`) and remove items from the start of the queue (`dequeue`). It also provides a function which suspends the running task on an empty queue until an item is placed into the queue, and then removes and returns the item (`waitForItem`). This class provides a generally accessible function which returns the number of items currently contained within the queue (`howMany`).

`Semaphore` - This class represents a resource lock or flag. Its implementation uses the underlying resource facilities of the RTX. This class provides functions which allocate the semaphore instance (`request`), and de-allocate the instance (`release`). It also provides a function which suspends the current task on an allocated instance until it is released (`waitFor`). It also provides a function which returns the current state of the instance (`retrieveStatus`).

`MemoryPool` - This class represents a pool of equally sized blocks of memory. It uses the underlying fixed-size memory management features of the underlying RTX. This class provides functions used by its subclasses to allocate a buffer from the pool (`allocate`) and release an allocated buffer back to the pool (`deallocate`). It also provides functions which suspend the current task on an empty pool until a buffer is released, and allocates the released buffer (`waitForBuffer`). It provides a general function which indicates the number of buffers currently available in the pool (`available`).

`TaskMonitor` - This class is a subclass of **`Task`** and is responsible for ensuring that none of the tasks within the instrument have crashed. This class operates by notifying each **`Task`** instance in the system and waiting for the task to reply (`respond`). Between each query, the TaskMonitor resets the watchdog timer (**`Watchdog`**). If a task does not respond within the watchdog time-out period (current ~8 minutes), the watchdog timer will expire and reset the instrument.

`SystemClock` - This class is responsible for maintaining the number of Back End Timer ticks since the last instrument reset. The constructor for this class starts the TimerDevice, and provides a function which supplies the current timer tick value (`currentTime`). NOTE: If the Back End Processor runs continuously, this counter will wrap about once every 13 years.

TimerCallback - This class is a subclass of ***Devices***::**DevCallback** and is responsible for passing control to ***Nucleus RTX*** during timer interrupt processing (invoke).

Devices**::Watchdog** - This class is a subclass of ***Devices***::**IntrDevice** and is provided by the ***Devices*** class category and is periodically reset by the **TaskMonitor** class. Refer to Section 7.0 for a description of this class

Devices**::Timer** - This class is a subclass of ***Devices***::**IntrDevice** and is provided by the ***Devices*** class category. It is used by the **SystemClock** class to provide system timer ticks to ***Nucleus RTX***. Refer to Section 7.0 for a detailed description of this class.

15.4 Interrupt Support and Task Event Definitions

15.4.1 Preemptive Context Switching

In order for ***Nucleus RTX*** to provide preemptive context switching, it must be called at the start of interrupt processing and must be allowed to control the return from the interrupt. Specifically, the **low-level interrupt handler** must call the ***Nucleus RTX*** function SKD_Interrupt_Context_Save prior to calling any C or C++ code. This function saves the state of all registers associated with the active task when the interrupt was taken. Once the **interrupt handler** is done, it must call SKD_Interrupt_Context_Restore. This function determines which task to run next, and restores the task's registers.

15.4.2 Task Notification Event Definitions

Tasks can suspend and resume their execution using event flags. Up to 32 flags can be waited on and sent to a particular task. These flags are specified as bit definitions in an unsigned integer value. Some of these flags are global, in that they may be sent to any task within the instrument, and other flags are privately used by a particular task. To avoid conflicting definitions, the lower 16-bits of an event word are used for these private task events, and the upper 16-bits are used for global events.The currently defined global event flag bit definitions are as follows (TBD):

```
enum EventPublic
{
    EV_TASKQUERY          = 1 << (0+16),// Task Monitor Query
    EV_SECOND_TICK        = 1 << (1+16),// 1 second tick (TBD)
    EV_STANDBY_ACTIVE     = 1 << (2+16),// Standby Signal went Active
    EV_STANDBY_INACTIVE = 1 << (3+16),// Standby Signal when Inactive
    EV_RADMON_ACTIVE      = 1 << (4+16),// Radiation Monitor went Active
    EV_RADMON_INACTIVE    = 1 << (5+16),// Radiation Monitor Inactive
    EV_RESERVED           = 1 << (6+16) // Reserved
}
```

15.5 Scenarios

15.5.1 Use 1: Launch Multiple Tasks

This section describes the overall initialization procedure used by ACIS to start-up the executive and launch the system's tasks. The configured task register and scheduling contexts are maintained within ***Nucleus RTX***, and are identified using task ids. Each **Task** class instance within ACIS contains a copy of this Nucleus id. The **TaskManager** maintains an array of pointers to **Task** instances, where the array is indexed using Nucleus task identifiers.

The system initialization requirements of the Real-time Executive system are complicated slightly by the nature of ***Nucleus RTX*** coupled with the initialization behavior of C++. The requirements are as follows:

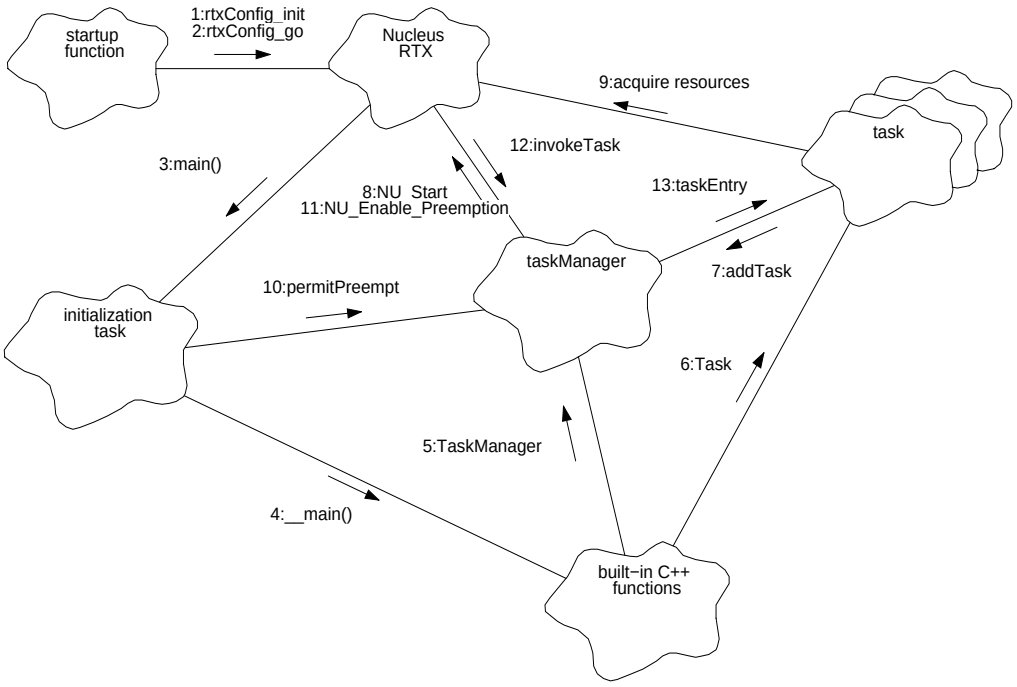
- ***Nucleus RTX*** requires that all of its configuration structures be initialized prior to starting the executive
- ***Nucleus RTX*** requires that access to its resources must wait until the executive has been started (i.e. don't try to use a queue until RTX is running).
- C++ invokes its global constructors (i.e. class initialization functions) as the first line of the standard C/C++ start-up function "main()."

In order to allow the executive classes to acquire RTX resources upon initialization, the software must either:

- In addition to a class's constructor, provide an "initialize()" function for each executive class, which must be invoked by someone after RTX has been started.
- Or, have the start-up code configure RTX to start only one initialization task, which then initializes all global class instances, using constructors, and starts the remaining tasks.

The ACIS software design uses the second alternative. Figure 51 illustrates the key elements of start-up scenario:

FIGURE 51. Executive Initialization Scenario



1. After performing the low-level processor initialization and patching, the system start-up function sets up the ***Nucleus RTX*** configuration data structures. This function installs all of the key ACIS tasks in an initial “Stopped” state. The entry point of each task is the *taskManager*’s `goInvokeTask`() function. It also installs a low-priority “initialization” task in an initial “Start” state. The initial task also comes up in a non-preemptive state, meaning that as tasks are started, they won’t take control from the initialization task until preemption is enabled.
2. The system start-up function tells ***RTX*** to initialize and to start executing.
3. ***RTX*** initializes and passes control to the only non-stopped task in the system, the low-priority initialization task.This task’s entry function then invokes the function `main`().
4. `main`() then invokes the built-in C++ function `__main`(), which proceeds to invoke the constructors of every global class instance in the system.
5. `__main`() invokes the constructor for the *taskManager* instance
6. `__main`() invokes the constructors for each of the task instances in the system
7. Each task instance constructor calls *taskManager*.`addTask`() to record the task in the manager’s data structures.
8. *taskManager* tells ***RTX*** to start the installed task.The task doesn’t actually run at this point because preemption is still disabled.
9. The task’s initialization code is now free to acquire whatever ***RTX*** resources it needs.
10. Once all of the constructors are invoked and `__main`() returns, `main`() finally tells *taskManager* to enable preemption.

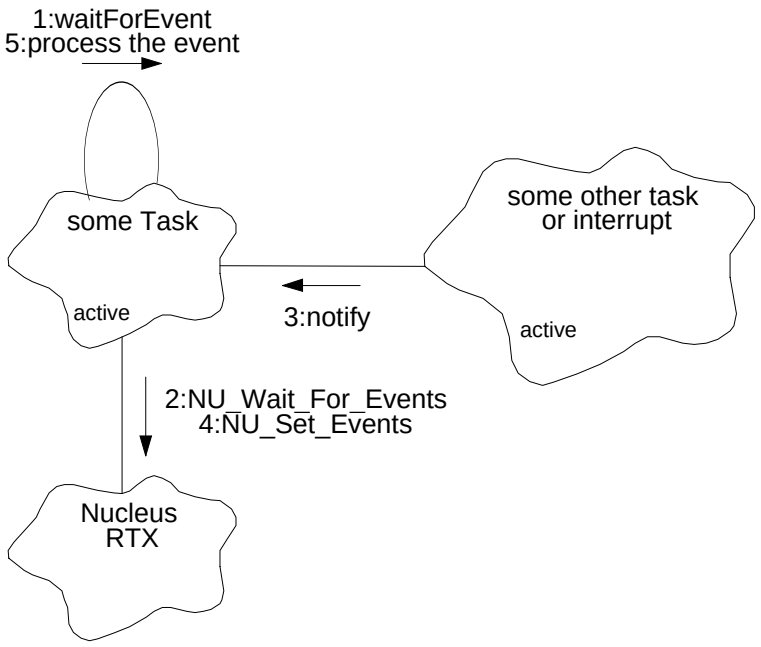
11. The *taskManager* then instructs ***RTX*** to enable preemption.
12. Once preemption is enabled, ***RTX*** invokes each of the installed tasks, in order of their priority. Higher priority tasks run until they suspend, and then the next lower priority task is invoked, etc.
13. *taskManager*'s `goInvokeTask`() function then invokes the active task instance's `goTaskEntry`() function and the task is now off and running.

15.5.2 Use 2: Notify tasks

This section briefly describes how individual tasks are notified of discrete events within the system.

Within ACIS, each **`Task`** instance has an associated dedicated ***`Nucleus RTX`*** Event Group (see Section 4.5). Whenever a task waits for an event, it only waits on events from this group. When a client notifies the task of the event, it invokes the task's `notify()` member function. This function then uses the underlying RTX to signal the task's Event Group, and reschedule the task. Whenever the task wakes up, it consumes the received event, and continues execution.

FIGURE 52. Task Notification



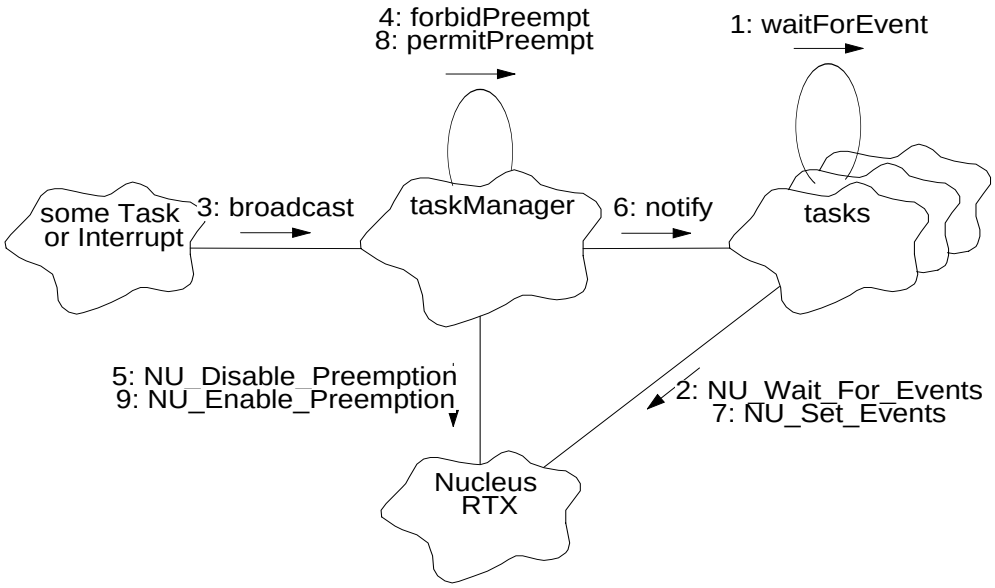
1. The task cannot proceed until some event is received, so it invokes `waitForEvent` to block until the event is received.
2. `waitForEvent` invokes the underlying `NU_Wait_For_Events` function, passing the task-specific *`event group id`*.
3. Some other task or interrupt handler signals the waiting task with the desired event, using `notify`.
4. `notify` then tells ***`RTX`*** to set the event(s) in the task's event group using `NU_Set_Events`
5. ***`RTX`*** returns control to the waiting task (appearing to return from `NU_Wait_For_Events` and `waitForEvent`). The task then proceeds to process the event.

15.5.3 Use 3: Broadcast events

This section briefly describes how a client broadcasts an event to all tasks in the system.

This feature uses the **`TaskManager`** to iterate through each task in the system, notifying each task of the event. To ensure that the task priorities are adhered to, the **`TaskManager`** disables preemption during this process.

FIGURE 53. Event Broadcasting



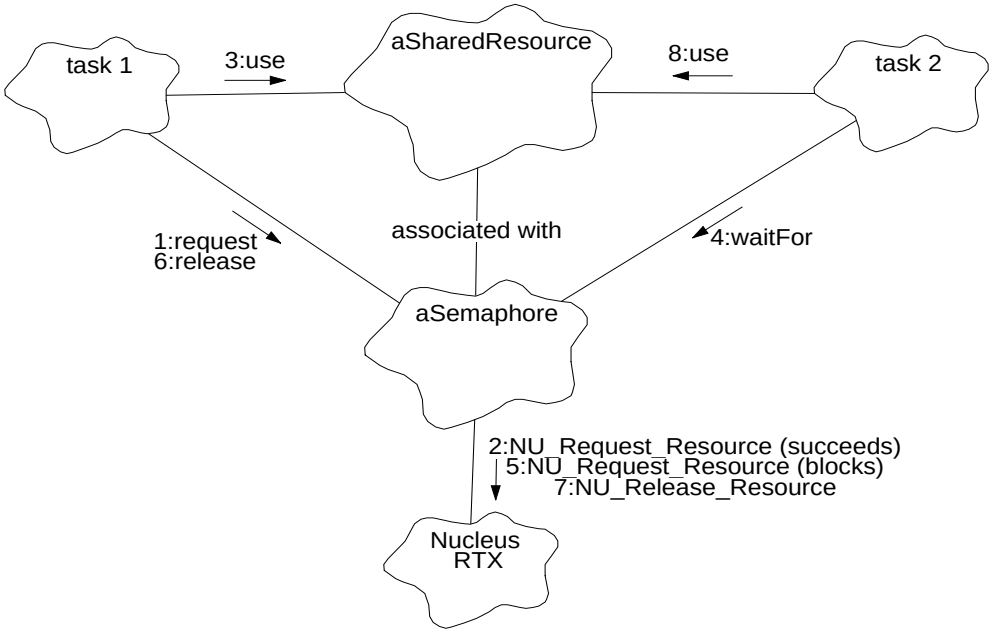
1. One or more tasks block for an event using `waitForEvent`
2. `waitForEvent` calls ***RTX***'s `NU_Wait_For_Events` to suspend the task
3. Some non-suspended task or interrupt handler tells the *`taskManager`* to `broadcast` an event to all tasks.
4. It then forbids preemption, using `forbidPreempt`
5. `forbidPreempt` calls the ***RTX*** function, `NU_Disable_Preemption` to prevent control from being stolen from the active task
6. The *`taskManager`* then calls `notify` for every task in the system.
7. Each task's `notify` function calls ***RTX***'s `NU_Set_Events` to signal the task
8. After all of the tasks have been notified, the *`taskManager`* invokes `permitPreempt`
9. `permitPreempt` uses the ***RTX*** `NU_Enable_Preemption` function to, once again, allow other tasks to take control from the one doing the notify. At this point, the tasks which were woken by the notify will take control, in the order of their relative priorities.

15.5.4 Use 4: Lock resources

This section briefly describes how resources are locked and unlocked by one or more clients.

The Executive uses a Semaphore Class to implement exclusive access locks. Each semaphore instance is associated with some sharable item within ACIS, and contains a ***Nucleus RTX*** resource identifier. Users of the sharable item must coordinate their use of the item using the associated Semaphore instance. In practice, access to a shared resource's semaphore may be hidden from the user by the member functions of the resource, but this is beyond the scope of this example. Figure 54 shows a scenario where one task obtains a resource's semaphore, causing another task to block until the semaphore is released. The memory required for the ***RTX*** resource is allocated by ***RTX*** during its start-up procedure.

FIGURE 54. Resource Arbitration



1. *task 1* gets the lock associated with the shared resource by invoking `request`
2. The semaphore's `request` function then invokes ***RTX***'s `NU_Request_Resource`, which succeeds and locks the associated ***RTX*** resource.
3. *task 1* proceeds to use the shared resource in relative safety
4. *task 2* wants access to the shared resource, and is willing to wait for it, so it invokes the `waitFor` function to attempt to obtain the resource, and, if necessary, block until it becomes available (or until it times out).
5. The `waitFor` function invokes ***RTX'***s `NU_Request_Resource` function again, this time passing in a time-out value. Since the resource is currently locked by *task 1*, `NU_Request_Resource` suspends *task 2*.

6. *task 1* finally finishes with the resource and unlocks it using `release`.
7. `release` invokes `NU_Release_Resource`. Once this happens, *task 2* is woken up, and its call to `NU_Request_Resource` returns, re-locking the resource for *task 2*'s benefit.
8. *task 2* then proceeds to use the resource, releasing when it's done (not shown).

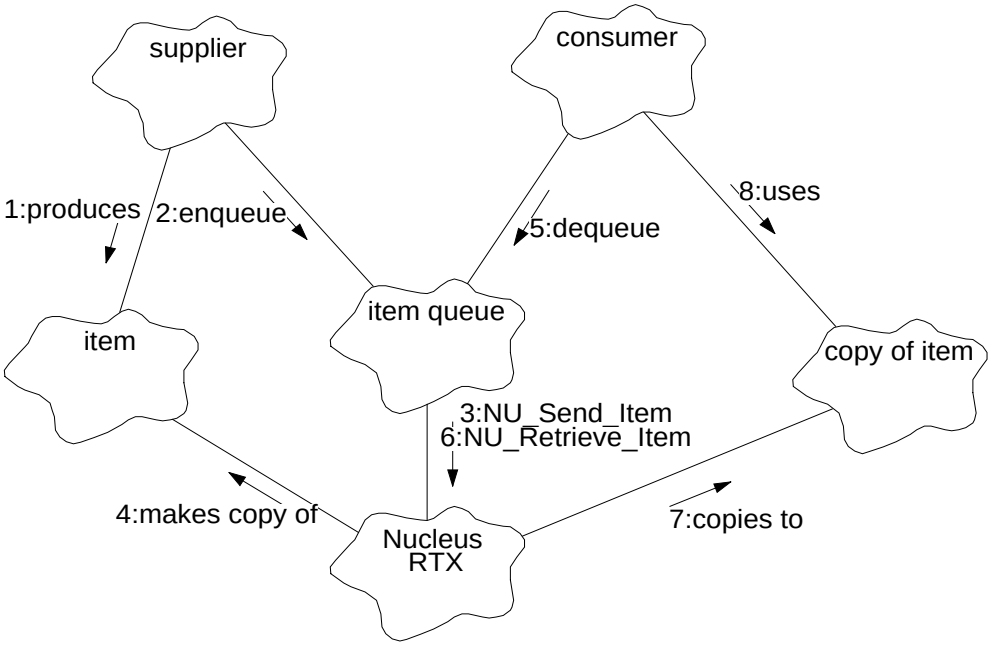
In general, the **`Semaphore`** class's `request` function is used for "balking requests", where, if the semaphore is already locked, the client does not want to block until the semaphore is released. If the client wants to wait until the semaphore is released, it should use the `waitFor` function. This function also provides a time-out feature. If the client is suspended longer than the time-out argument passed to `waitFor`, `waitFor` will fail to obtain the lock on the semaphore.

15.5.5 Use 5: Maintain queues

This section briefly describes a scenario where two clients are using a queue to pass information between them.

ACIS uses a Queue class to manage first-in, first-out inter-process queues of information. Each instance of a queue is associated with an Nucleus RTX queue. The memory required for the ***RTX*** queue is allocated by ***RTX*** during its start-up procedure.

FIGURE 55. Simple Queue Use Example



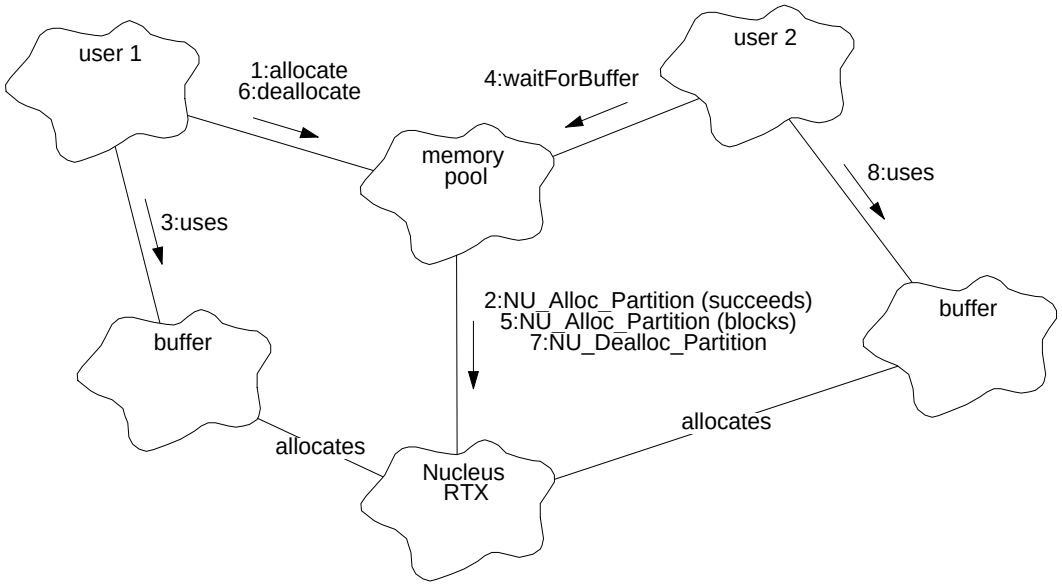
1. A supplier produces an item
2. and calls `enqueue` to copy the item into the item queue, passing the address of the item
3. The **Queue** instance, item queue, invokes the ***RTX*** `NU_Send_Item` function to place the item at the end of the queue
4. `NU_Send_Item` copies the data to the end of the queue
5. The consumer invokes `dequeue`, passing a pointer to a buffer to store the item
6. `dequeue` invokes the ***RTX*** function `NU_Retrieve_Item` to copy the next item from the start of the queue, and remove the copied item from the queue
7. The consumer then uses the retrieved item.

15.5.6 Use 6: Acquire and release memory

This section briefly describes how memory is allocated and released from a pool of buffers.

The executive category provides a **`MemoryPool`** class which manages pools of equally sized buffers. Each **`MemoryPool`** instance refers to a unique RTX memory partition. The memory for the pool is allocated by RTX during its start-up procedure.

FIGURE 56. Simple Memory Pool Use



1. user 1 calls **`MemoryPool`**'s `allocate` to obtain a buffer.
2. `allocate` invokes the ***RTX*** function `NU_Alloc_Partition` to obtain a buffer from the pool. For this example, assume that there is only one buffer in the pool, and so after the `NU_Alloc_Partition` call, the pool is empty.
3. user 1 successfully gets the last buffer in the pool and starts using it
4. user 2 attempts to get a buffer, blocking if necessary, using `waitForBuffer`.
5. `waitForBuffer` calls `NU_Alloc_Partition`. Since the pool is empty, `NU_Alloc_Partition` blocks until a buffer is released (or until it time-out)
6. user 1 finally finishes with the buffer and calls `deallocate` to release it
7. `deallocate` invokes ***RTX***'s `NU_Dealloc_Partition` to release the buffer. Once the buffer is released, user 2's call to `waitForBuffer`/`NU_Alloc_Partition` succeeds and returns with the allocated buffer.
8. user 2 is now free to use the obtained buffer

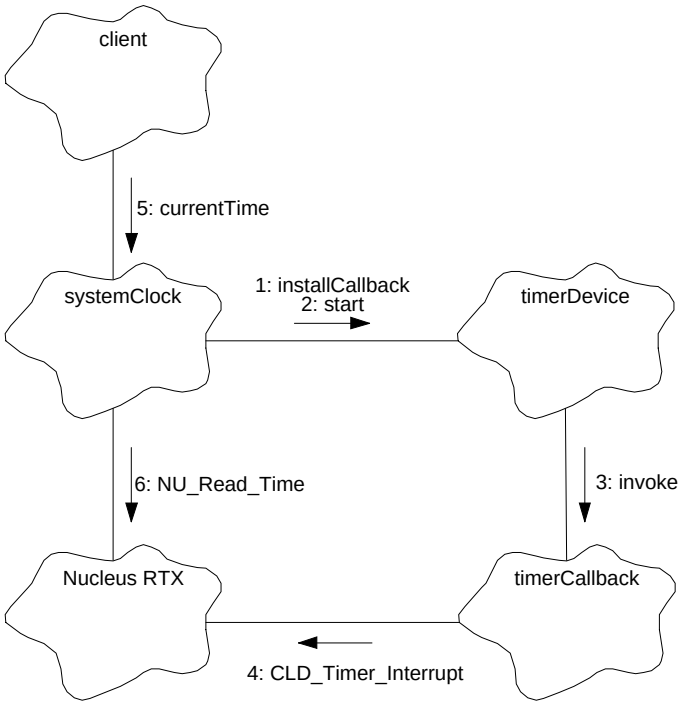
15.5.7 Use 7: Provide current time

Nucleus RTX maintains an internal timer tick counter, which it increments for each call to `CLD_Timer_Interrupt`. This function, `CLD_Timer_Interrupt`, also performs all of the executive's time-based scheduling operations, such as suspension time-outs.

The Executive interface uses the **`SystemClock`** class to install an interrupt callback instance into the **`TimerDevice`**, and to access the Nucleus timer tick counter.

Figure 57 illustrates timer interrupt handling, and how clients access the Nucleus timer tick counter value.

FIGURE 57. Executive Timer Interrupt Handling and Counter Access



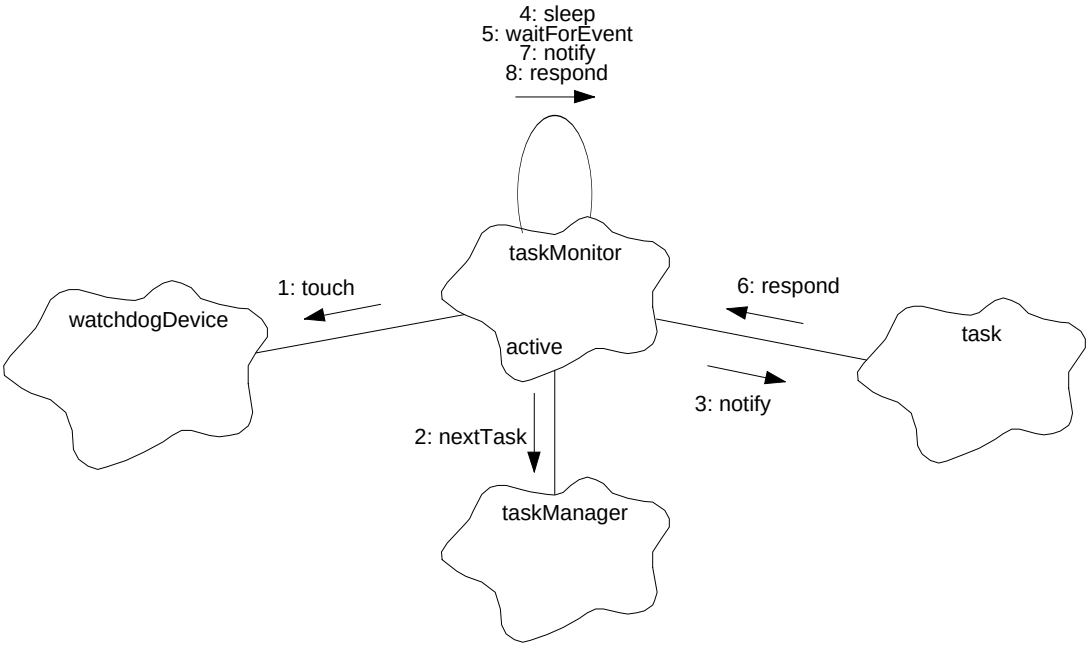
1. During system initialization, the *`systemClock`* installs the *`timerCallback`* instance into the *`timerDevice`*, using *`timerDevice`*.`installCallback`().
2. *`systemClock`* then starts the timer, using *`timerDevice`*.`start`().
3. Later, once the timer hardware count expires, a timer interrupt is taken. As part of its interrupt processing, the *`timerDevice`* invokes the installed callback, *`timerCallback`*.invoke().
4. *`timerCallback`*.`invoke`() calls ***Nucleus RTX*** function, `CLD_Timer_Interrupt`(), to count the tick, and schedule its tasks.
5. Clients obtain the current timer tick counter by calling *`systemClock`*.`currentTime`().
6. `currentTime`() then calls the ***Nucleus RTX*** function, `NU_Read_Time`), to obtain its timer tick counter.

15.5.8 Use 8: Verify task health

In order to ensure that no task has crashed and entered an infinite loop, the executive implements a utility task, **`TaskMonitor`**, which periodically polls each task in the system, and resets the watchdog timer between each query. If a task does not respond within the time-out period of the watchdog timer, the instrument is reset by the timer hardware.

Figure 58 illustrates the how the **`TaskMonitor`** behaves in the system.

FIGURE 58. TaskMonitor queries and responses



1. After system initialization, the executive calls the main function of the monitor's task, `goTaskEntry`(). This function sets the current task pointer to 0, and enters an infinite loop. At the top of the loop, the *`taskMonitor`* resets the watchdog timer, using *`watchdogDevice`*.`touch`().
2. The *`taskMonitor`* then queries the *`taskManager`* for the next task in its list, passing *`taskManager`*.`nextTask`() the current task pointer.
3. *`taskMonitor`* then notifies the referenced task using *`task`*->`notify`().
4. *`taskMonitor`* then sleeps for a TBD period of time, using `sleep`, to space out the queries and allow lower priority tasks to run.
5. Once sleep returns, the *`taskMonitor`* suspends until the queried task responds, using `waitForEvent`().
6. Once the *`task`* receives the query, it replies to the monitor using the monitor's binding function, *`taskMonitor`*.`respond`().
7. `respond`() wakes up the task portion of the *`taskMonitor`* using `notify`(). At this point, the *`taskMonitor`* touches the watchdog, and repeats from step 2.

8. When the *taskMonitor* sends itself a query, its call to `waitForEvent`() (step 5) will return indicating the query. In this case, the *taskMonitor* calls its `respond`() function which then notifies the *task*. The subsequent call to `waitForEvent`() will return with the response immediately, and the *taskMonitor* proceeds with the next task.

15.6 Class TaskManager

Documentation:

The **`TaskManager`** is responsible for co-ordination the execution of tasks within the executive. It supplies operations which apply globally to all tasks.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Uses:

`Task`

Public Interface:

Operations:
```
TaskManager()
addTask()
broadcast()
forbidPreempt()
goInvokeTask()
nextTask()
permitPreempt()
queryCurrentTask()
```

Protected Interface:

Has-A Relationships:

`Task*` *`taskList`*[]: This is an array of pointers to tasks. The array is indexed by the ***`Nucleus RTX`*** Task Identifier associated with a given task.

Concurrency: Synchronous

15.6.1 TaskManager()

Public member of: **`TaskManager`**

Documentation:

This is the constructor for the **`TaskManager`**. It initializes all state variables.

Preconditions:

`Nucleus RTX` is running, but only the current task is active.

Semantics:

Flag preemption as disabled

Zero the array of task pointers.

Postconditions:

The **`TaskManager`** is ready to accept `addTask` requests and, in general, ready to go.

Concurrency: Sequential

15.6.2 addTask()

Public member of: **TaskManager**

Return Class: **void**

Arguments:

Task* *task*
unsigned *taskId*

Documentation:

This function instructs the **TaskManager** to add *task* to its list of tasks, and to associate the task with the RTX identifier, *taskId*.

This function is intended to be used by the **Task** class's constructor to register the task with the **TaskManager**.

Preconditions:

Preemption of the currently running task must be disabled

taskId must be an existing Nucleus RTX task index

A *task* using *taskId* must not have already been installed

task must not be NULL

Semantics:

Install *task* in the Task array entry indexed by *taskId* slot.

Tell ***Nucleus RTX*** to start the task (preemption disabled so task won't run right away)

Postconditions:

task is installed in the **TaskManager**'s array and is ready to run once preemption is enabled and all higher priority tasks suspend.

Concurrency: Sequential (note the preemption requirement)

15.6.3 broadcast()

Public member of: **`TaskManager`**

Return Class: **`void`**

Arguments:

`unsigned` *`eventMask`*

Documentation:

Notify all tasks of the events indicated in *`eventMask`*

Preconditions:

None

Semantics:

Forbid preemption, notify all tasks of *`eventMask`*, and restore preemption

Postconditions:

All tasks in the system will have been notified

Concurrency: Synchronous

15.6.4 forbidPreempt()

Public member of: **`TaskManager`**

Return Class: **`void`**

Documentation:

Prevents the current task from being preempted. Within a given task, subsequent calls to `forbidPreempt`() prior to `permitPreempt`() accumulate. An equal number of calls to `permitPreempt`() are needed to enable preemption.

NOTE: THIS DOES NOT DISABLE INTERRUPTS!!!

Preconditions:

Must be at task level (not at interrupt level)

Semantics:

Prevent preemptive context switch, call `NU_Disable_Preemption`

Postconditions:

Preemption is disabled

Concurrency: Synchronous

15.6.5 goInvokeTask()

Public member of: **`TaskManager`**

Return Class: **`void`**

Documentation:

This function invokes the `goTaskEntry`() function of the **`Task`** instance associated with the currently running ***RTX*** task.

Preconditions:

Nucleus RTX is running, and the active task has been installed using `add-Task`.

Semantics:

Get the current task id from ***RTX*** using `NU_Current_Task`, use as index into task table and return the referenced **`Task`** pointer and calls its `goTaskEntry`() member function.

Concurrency: Synchronous

15.6.6 nextTask()

Public member of: **TaskManager**

Return Class: **Task***

Arguments:

Task* *oldTask*

Documentation:

This function allows clients to iterate through each task in the system. If *oldTask* is NULL, this function returns the first task in the system. If it is not NULL, it returns a pointer to the next task.

Preconditions:

oldTask is either NULL or a pointer to a **Task** installed via *addTask*.

Semantics:

If *oldTask* is NULL, retrieve the **Task** pointer indexed by 0.

If *oldTask* is not NULL, retrieve the id of *oldTask*, and add 1. If result greater than or equal to the number of installed tasks, wrap the index to 0, and get the first **Task**. If result is a valid index, get pointer to indexed **Task**.

Postconditions:

Returns pointer to the “next” **Task** instance

Concurrency: Synchronous

15.6.7 permitPreempt()

Public member of: **`TaskManager`**

Return Class: **`void`**

Documentation:

Allow the current task to be preempted. Within a given task, one call to `permitPreempt`() for each outstanding call to `forbidPreempt`() is needed to enable preemption. Additional calls to `permitPreempt`() have no effect.

NOTE: THIS DOES NOT ENABLE INTERRUPTS!!!

Preconditions:

Must not be in an interrupt handler

Semantics:

Allow task to be preempted by another higher priority task.

Postconditions:

Current task may now be preempted by another

Concurrency: Synchronous

15.6.8 queryCurrentTask()

Public member of: **`TaskManager`**

Return Class: **`Task*`**

Documentation:

Returns a pointer to the currently active task. If this function is called as part of an interrupt handler, it returns a pointer to the task that was interrupted. If all tasks were suspended when the interrupt is taken, this function returns 0.

Preconditions:

None

Semantics:

Get the currently active task

Postconditions:

Return value points to the active task

Concurrency: Synchronous

15.7 Class Task

Documentation:

The **`Task`** Class defines a thread of control within the system.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
Task()
getTaskId()
goTaskEntry()
notify()
requestEvent()
sleep()
waitForEvent()
yield()
```

Protected Interface:

Has-A Relationships:

unsigned *`rtxTaskId`*: This variable contains the ***`Nucleus RTX`*** Task Identifier associated with this instance of a task.

unsigned *`rtxEventGroupId`*: This variable contains the ***`Nucleus RTX`*** Event Group identifier associated with the event group dedicated to this task instance.

Concurrency: Active

15.7.1 Task()

Public member of: **Task**

Arguments:

unsigned *taskId*

Documentation:

This initializes an instance of **Task**. *taskId* is the ***Nucleus RTX*** task identifier associated with the instance.

Preconditions:

Nucleus RTX must be already running. *taskManager* must exist and must have been initialized (constructed).

Semantics:

Store *taskId* in both *rtxTaskId* and using for *rtxEventGroupId*. Call *taskManager*.addTask() to register the task with the manager.

Postconditions:

The task is installed and ready to run.

Concurrency: Sequential

15.7.2 getTaskId()

Public member of: **`Task`**

Return Class: **`unsigned`**

Documentation:

This function returns a numeric task identifier. This function is intended for use only by the **`TaskManager`**.

Preconditions:

None

Semantics:

Get *`rtxTaskId`*

Postconditions:

Return *`rtxTaskId`*

Concurrency: Synchronous

15.7.3 goTaskEntry()

Public member of: **`Task`**

Return Class: **`void`**

Documentation:

This function is an abstract function which represents the main entry point of a task. All subclasses of **`Task`** must implement a version of this function.

Preconditions:

The task must not have already been started.

Semantics:

Subclasses of task must implement this function. In general, it contains the main loop of the task.

Postconditions:

THIS FUNCTION MUST NEVER RETURN.

Concurrency: Synchronous

15.7.4 notify()

Public member of: **`Task`**

Return Class: **`void`**

Arguments:

`unsigned` *`eventMask`*

Documentation:

This function notifies the task that the events specified in *`eventMask`* bit-field have occurred.

Preconditions:

None

Semantics:

Set the events in the event group specified for the task using `NU_Set_events`. Use *`rtxEventGroupId`* as the notification group.

Postconditions:

Adds the events specified *`eventMask`* to current pending events and causes task to resume if it was blocked on one or more of the events

Concurrency: Synchronous

15.7.5 requestEvent()

Public member of: **`Task`**

Return Class: **`unsigned`**

Arguments:

`unsigned` *`eventMask`*

Documentation:

This function tests to see if one or more events specified in *`eventMask`* are set, and if so, consumes and returns those set events. Events not specified in *`eventMask`* are not affected. Unlike `waitForEvent`, this function does not block.

Preconditions:

None

Semantics:

Use `NU_Get_Events` with the NU_NO_TIMEOUT argument to cause it to return immediately.

Postconditions:

Consumes any pending events which were active and specified in *`eventMask`*. It returns a mask containing the events which were consumed.

Concurrency: Synchronous

15.7.6 sleep()

Public member of: **`Task`**

Return Class: **`void`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function suspends the task for at least *`timeout`* number of timer ticks (1/10 second per tick).

Preconditions:

None

Semantics:

Use `NU_Sleep` to suspend the task

Postconditions:

Once timeout is reached, the task will be scheduled to run.

Concurrency: Synchronous

15.7.7 waitForEvent()

Public member of: **`Task`**

Return Class: **`unsigned`**

Arguments:

`unsigned` *`eventMask`*

Documentation:

This function causes the task to suspend execution until one or more of the events ids in *`eventMask`* is set (via a **`Task`**::`notify`() or **`TaskManager`**::`broadcast`()). The returned *`eventMask`* contains the events which were waited for that were notified.

Preconditions:

None

Semantics:

Suspends task execution until one or more event ids in *`eventMask`* is set. Use `NU_Wait_For_Events` with NU_WAIT_FOREVER as the argument.

Postconditions:

Resumes execution

Returns the events which were satisfied

Clears the waited for/received events pending_events

Concurrency: Synchronous

15.7.8 yield()

Public member of: **`Task`**

Return Class: **`void`**

Documentation:

This function allows other tasks of the same priority level to run. This function will not return until at least all other tasks of the same priority have either suspended, or yielded.

Preconditions:

None

Semantics:

Use `NU_Relinquish` to give up control

Postconditions:

Once all other tasks of the same priority have suspended or yielded, this function will return.

Concurrency: Synchronous

15.8 Class SystemClock

Documentation:

This class represents the executive's system clock.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Implementation Uses:

Timer
TimerCallback

Public Interface:

Operations: `currentTime()`

Concurrency: Synchronous

Persistence: Persistent

15.8.1 currentTime()

Public member of: **SystemClock**

Return Class: **unsigned**

Documentation:

This function returns the number of timer-ticks (1/10 second units) since the point at which interrupts were first enabled after start-up.

Semantics:

Call `NU_Read_Time` to get the current timer tick count value.

Concurrency: Synchronous

15.9 Class TimerCallback

Documentation:

This class is responsible for passing control to Nucleus RTX during timer tick interrupt handling.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`DevCallback`**

Public Interface:

Operations: `invoke()`

Concurrency: Synchronous

Persistence: Persistent

15.9.1 invoke()

Public member of: **`TimerCallback`**

Return Class: **`void`**

Arguments:

`IntrDevice*` *`devptr`*

Documentation:

This function is invoked by the timer device interrupt handler, and is responsible for passing control to Nucleus RTX. *`devptr`* points to the device issuing the callback. This function calls the Nucleus `CLD_Timer_Interrupt`() to maintain its timer tick counter, and perform any time-based context switch preparation.

Concurrency: Guarded

15.10 Class TaskMonitor

Documentation:

The Task Monitor is responsible for testing each task in the system, including itself, and resetting the watchdog timer as each tested task responds.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Implementation Uses:

`Watchdog`
`TaskManager`

Public Interface:

Operations:
```
TaskMonitor()
goTaskEntry()
respond()
```

Private Interface:

Has-A Relationships:

`Task*` *`curquery`*: This is a pointer to the task instance currently being queried by the task monitor. If no query is outstanding, this pointer is 0.

`const unsigned` *`delay`*: This is the number of timer ticks to sleep between task queries. Its value is TBD.

Concurrency: Active

Persistence: Persistent

15.10.1 TaskMonitor()

Public member of: **`TaskMonitor`**

Arguments:

`unsigned` *`taskid`*

Documentation:

This function is the task's constructor. *`taskid`* is the ***`Nucleus RTX`*** identifier associated with the task.

Concurrency: Sequential

15.10.2 goTaskEntry()

Public member of: **`TaskMonitor`**

Return Class: **`void`**

Documentation:

This is the main entry point for the task monitor. This function consists of an infinite loop, which queries the **`TaskManager`** for each task, and then queries each task. As each task responds, this function resets the watchdog timer.

Semantics:

Zero *`curquery`* and enter the infinite loop. Touch the *`watchdogDevice`*. Pass *`curquery`* to *`taskManager`*.`nextTask`() and store in *`curquery`*. Query the task using `notify`(). sleep for *`delay`* ticks. Wait for either the responds, or a query. If query, call `respond`() and try again. Once response is received, repeat infinite loop.

Concurrency: Synchronous

15.10.3 respond()

Public member of: **`TaskMonitor`**

Return Class: **`void`**

Documentation:

This function is used by the system's tasks (including itself) to respond to queries from the task monitor.

Semantics:

`notify` ***`taskMonitor`*** that the queried task is alive.

Concurrency: Synchronous

15.11 Class Semaphore

Documentation:

This class is used to provide a mechanism by which several tasks can coordinate their use of a shareable resource.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
Semaphore()
release()
request()
retrieveStatus()
waitFor()
```

Protected Interface:

Has-A Relationships:

unsigned *rtxSemId*: This variable contains the ***Nucleus RTX*** Resource Id associated with this instance of a semaphore.

Concurrency: Synchronous

15.11.1 Semaphore()

Public member of: **Semaphore**

Arguments:

unsigned *semId*

Documentation:

This is the constructor for the class. *semId* is the ***RTX*** resource id to associate with this instance.

Preconditions:

Nucleus RTX must be running

Semantics:

Store *semId* in *rtxSemId*

Postconditions:

The **Semaphore** is ready to be used.

Concurrency: Sequential

15.11.2 release()

Public member of: **`Semaphore`**

Return Class: **`void`**

Documentation:

This function unlocks a semaphore, indicating that the associated resource is available for use.

Preconditions:

Resource must already have been locked

Semantics:

Call `NU_Release_Resource` to indicate that the resource is available.

Postconditions:

Resource must now be unlocked and available. Any tasks which blocked on the resource are now unblocked (but the highest priority task will then get control and re-lock it).

Concurrency: Synchronous

15.11.3 request()

Public member of: **`Semaphore`**

Return Class: **`Boolean`**

Documentation:

This function provides a non-blocking way of obtaining ownership of a resource. It returns *`BoolTrue`* if the semaphore was successfully obtained and locked, and *`BoolFalse`* if someone else has already locked it.

Preconditions:

None

Semantics:

Call `NU_Request_Resource` passing NU_NO_TIMEOUT to prevent blocking.

Postconditions:

If lock is available, lock it and return *`BoolTrue`*

If lock is unavailable, return *`BoolFalse`*

Concurrency: Synchronous

15.11.4 retrieveStatus()

Public member of: **`Semaphore`**

Return Class: **`Boolean`**

Documentation:

This function retrieves the status of the semaphore. If locked it returns *`BoolTrue`*, otherwise it returns *`BoolFalse`*.

Preconditions:

None

Semantics:

Use `NU_Retrieve_Resource_Status`. Examine number tasks on the resource. 0 tasks means no-one's got it locked.

NOTE: Be careful here. If you get preempted between the query and testing the result, the answer may not be right (hence the Guarded concurrency qualifier on the function).

Postconditions:

If locked, return *`BoolTrue`*

If unlocked, return *`BoolFalse`*

Concurrency: Guarded

15.11.5 waitFor()

Public member of: **`Semaphore`**

Return Class: **`Boolean`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function waits for and obtains a semaphore. If successful, it returns *`BoolTrue`*. If *`timeout`*, expressed in timer ticks (1/10 second) expires, it returns *`BoolFalse`*.

Preconditions:

None

Semantics:

Use `NU_Request_Resource`, using *`timeout`* to bound the blocking time.

Postconditions:

If the lock becomes available, it locks it and returns *`BoolTrue`*.

If *`timeout`* expires before the lock is released, it returns *`BoolFalse`*.

Concurrency: Synchronous

15.12 Class Queue

Documentation:

The Queue class serves to encapsulate the inter-process queueing facilities of the RTX. It provides the underlying mechanisms to add, retrieve and query queue items. (NOTE: The `dequeue`, `enqueue` and `waitForItem` functions are not type-safe. Subclasses must provide type-safe wrappers around these functions)

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
Queue()
howMany()
```

Protected Interface:

Operations:
```
dequeue()
enqueue()
waitForItem()
```

Has-A Relationships:

unsigned *rtxQueueId*: This variable holds the ***Nucleus RTX*** Queue identifier associated with this instance of a queue.

Concurrency: Synchronous

15.12.1 Queue()

Public member of: **Queue**

Arguments:

unsigned *queueId*
unsigned *itemSize*
unsigned *itemCnt*

Documentation:

This is the constructor for the queue class. *queueId* identifies which ***RTX*** queue to use. *itemSize* and *itemCnt* are used to ensure that the queue's parameters match those assumed by the underlying RTX queue instance. *itemSize* is the maximum number of 32-bit words for any item placed into the queue, and *itemCnt* is the maximum number of items that can be held by the queue.

Preconditions:

Nucleus RTX must be running

The RTX queue identified by *queueId* must hold *itemCnt* elements whose size is *itemSize* number of words.

Semantics:

Check the queue count and item size against the reference RTX queue. Store the *queueId* in *rtxQueueId*

Postconditions:

The queue is ready for use.

Concurrency: Sequential

15.12.2 dequeue()

Protected member of: **Queue**

Return Class: **Boolean**

Arguments:

void* *item*

Documentation:

This function copies the first element from the queue into the buffer pointed to by *item*, and removes the element from the queue. If no elements are in the queue, the function immediately returns *BoolFalse*. If the element was successfully copied, it returns *BoolTrue*.

Preconditions:

The calling subclass MUST ensure that *item* points to a buffer large enough to hold the size for this element.

Semantics:

Dequeue item into specified buffer using NU_Retrieve_Item

Postconditions:

Buffer pointed to by *item* contains first element

The first element has been removed from the queue

Concurrency: Synchronous

15.12.3 enqueue()

Protected member of: **Queue**

Return Class: **Boolean**

Arguments:

const void* *item*

Documentation:

Enqueues *item* into this instances ***RTX*** queue. If the queue is full, the function returns *BoolFalse*, and nothing will be enqueued. If the queue has room, the item is copied, and the function returns *BoolTrue*.

Preconditions:

Item must point to buffer containing element whose size is less than or equal to the queue item size.

Semantics:

Copy data pointed to by *item* into the queue, using NU_Send_Item.

Postconditions:

If not full, *item*'s data is appended to the end of the queue and the function returns *BoolTrue*. If full, nothing copied and function returns *BoolFalse.*

Concurrency: Synchronous

15.12.4 howMany()

Public member of: **`Queue`**

Return Class: **`unsigned`**

Documentation:

Returns the number of items in the queue.

Preconditions:

None

Semantics:

Retrieve the number of items in the queue

NOTE: You may get preempted and someone enqueues or dequeues an element before you act on the return value (hence the Guarded concurrency qualifier).

Postconditions:

Returns the number of enqueued items

Concurrency: Guarded

15.12.5 waitForItem()

Protected member of: **Queue**

Return Class: **Boolean**

Arguments:

void* *item*
unsigned *timeout*

Documentation:

This function attempts to dequeue an element into *item*. If an element is present, or one is enqueued prior to timeout, *item* will be filled with a copy of the element and the function returns *BoolTrue*. If *timeout*, expressed in timer ticks (1/10 seconds), expires, *item* is left alone and the function returns *BoolFalse*.

Concurrency: Synchronous

15.13 Class MemoryPool

Documentation:

This class manages a collection of equally sized buffers.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
MemoryPool()
available()
```

Protected Interface:

Operations:
```
allocate()
deallocate()
waitForBuffer()
```

Has-A Relationships:

`unsigned` *`rtxPoolId`*: This variable holds the ***`Nucleus RTX`*** Memory Pool identifier.

Concurrency: Synchronous

15.13.1 MemoryPool()

Public member of: **MemoryPool**

Arguments:

unsigned *poolId*
unsigned *bufsize*
unsigned *nbufs*

Documentation:

This is the constructor for the class. *poolId* identifies which ***RTX*** memory partition to use for this instance. *bufsize* and *nbufs* are used to ensure that the pool's parameters match those assumed by the underlying RTX memory pool instance. *bufsize* is the maximum number of 32-bit words for any buffer contained within the pool, and *nbufs* is the maximum number of buffers maintained by the pool.

Preconditions:

Nucleus RTX must be running

The RTX memory pool identified by *poolId* must hold *nbufs* buffers whose size is *bufsize* number of words.

Semantics:

Verify pool buffer size and buffer count. Copy *poolId* to *rtxPoolId*

Postconditions:

The memory pool is ready for use.

Concurrency: Sequential

15.13.2 allocate()

Protected member of: **`MemoryPool`**

Return Class: **`void *`**

Documentation:

Attempts to allocate one buffer from this pool. Returns pointer to the buffer on success. Returns 0 if no buffers are available.

Preconditions:

None

Semantics:

Attempt to grab a buffer from the pool using `NU_Alloc_Partition`

Postconditions:

If buffer is available, decrease space remaining and return ptr to buffer.

If nothing is available, return 0

Concurrency: Synchronous

15.13.3 available()

Public member of: **`MemoryPool`**

Return Class: **`unsigned`**

Documentation:

Return number of blocks available in this Pool.

Preconditions:

None

Semantics:

Obtain the available number of blocks using
`NU_Available_Partitions`

NOTE: You may be preempted before you use the result, and the higher priority task may allocate or release a buffer, changing the true answer (hence the Guarded concurrency qualifier).

Postconditions:

If pool is empty, return 0 If no blocks are allocated, return maximum number of blocks. If some blocks are allocated, return number of remaining

Concurrency: Guarded

15.13.4 deallocate()

Protected member of: **MemoryPool**

Return Class: **void**

Arguments:

void* *block*

Documentation:

Return Memory Block, pointed to by *block*, to the pool.

Preconditions:

block must have been allocated from this pool

Semantics:

Release *block* back to pool using NU_Dealloc_Partition.

Postconditions:

Block is released to pool, available space increases

If tasks were waiting on the empty pool, they are unblocked.

Concurrency: Synchronous

15.13.5 waitForBuffer()

Protected member of: **`MemoryPool`**

Return Class: **`void*`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function attempts to allocate a buffer from the pool, and blocks until one is available, or until *`timeout`*, specified in timer ticks (1/10 second) is reached. If it succeeds, it returns a non-NULL pointer to the block. If it times out, it returns NULL.

Preconditions:

None

Semantics:

Call `NU_Alloc_Partition`, passing *`timeout`*. If no buffer is available, the task is suspended until a buffer is released by another task, or until *`timeout`* is reached. If a buffer becomes available, allocated and return with its pointer.

Postconditions:

If buffer is or becomes available prior to timeout, allocated it and return its pointer. If timeout is reached, return 0.

Concurrency: Synchronous

16.0 Command Management Classes (36-53213 A+)

16.1 Purpose

The purpose of the Command Management system is to receive, execute and log software commands received by the instrument software.

16.2 Uses

Use 1:: Acquire, execute and echo commands
Use 2:: Handle command device and protocol errors

16.3 Organization

Figure 59 illustrates the top-level classes and their relationships involved in processing commands, and Figure 60 illustrates the relationships used to echo command to telemetry.

FIGURE 59. Command Management Class Relationships

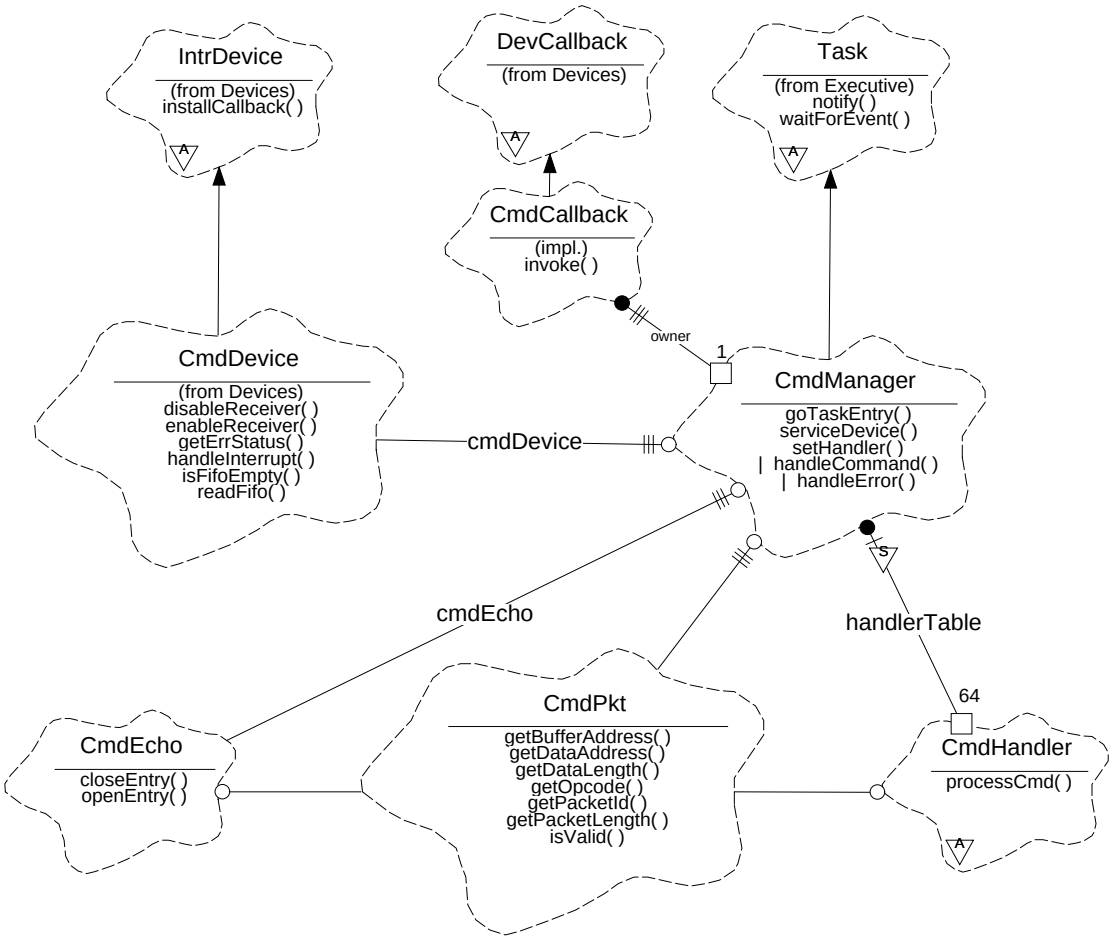
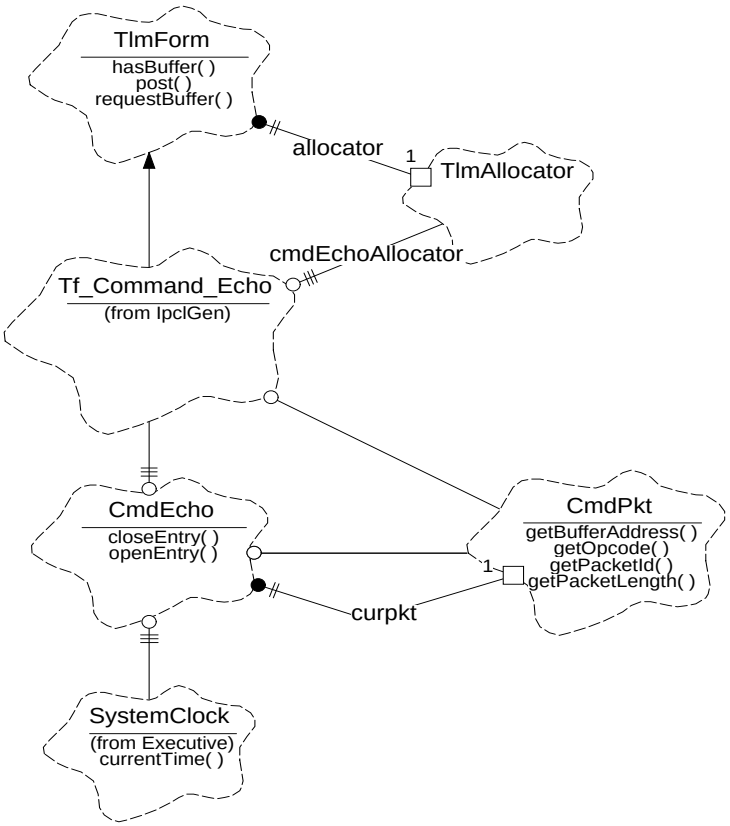


FIGURE 60. Command Echo class relationships



CmdManager- The **CmdManager** class is a subclass of ***Executive***::**Task** (see Section 15.0). There is only one instance of this class, called *cmdManager*. This class is responsible for actively waiting for commands or errors from the command device, and for executing the commands, and recovering from errors. This class uses the ***Devices***::**CmdDevice** class to provide an interface to the command hardware, the **CmdCallback** class to install an interrupt callback in the **CmdDevice**, the **CmdLog** class to log and echo processed commands, and the **CmdPkt** class to contain the body of a command packet. The **CmdManager** also contains a table of 64 pointers to subclass of **CmdHandler**. It uses the *opcode* field of a given command packet to lookup the corresponding handler from this table.

CmdHandler- This class is an abstract class which represents an object responsible for executing a particular command or group of commands. Every subclass of **CmdHandler** is responsible for implementing a command-specific `processCmd`() member function. This function verifies the command arguments, rejecting illegal commands, and processing accepted commands. Redundant command argument checking by other classes should be minimized.

CmdPkt- This class is used to hold the contents of a command packet, and to provide access to the packet's header information, and to its data.

CmdCallback- This class is a subclass of ***Devices***::**DevCallback**. It is used by the **CmdManager** to obtain control during processing of command interrupts.

CmdEcho- This class is responsible for acknowledging the receipt and execution of commands. It provides functions which obtain and log the approximate arrival time of the packet (`openEntry`), and which forms and telemeters the echo of the command, along with its arrival time and its disposition (`closeEntry`).

Tf_Command_Echo - This class is a subclass of **TlmForm**, and is generated from the IP&CL. It is responsible for formatting a telemetry packet containing a copy of a command, its arrival time, and disposition code.

TlmForm - This class is responsible for formatting and posting telemetry packet buffers, and is described in Section 18.0.

TlmAllocator - This class is responsible for maintaining pools of telemetry packet buffers, and is described in Section 18.0.

SystemClock - This class is defined by the ***Executive*** class category, and is described in Section 15.0.

Task - This class is defined by the ***Executive*** class category, and is described in Section 15.0.

CmdDevice - This class is a subclass of **IntrDevice**, and is defined by the ***Devices*** class category, and is described in Section 8.0.

IntrDevice - This class is defined by the ***Devices*** class category, and is described in Section 6.0.

DevCallback - This class is defined by the ***Devices*** class category, and is described in Section 6.0.

16.4 Command Processing Assumptions and Restrictions

16.4.1 Packet Format

The ACIS instrument software receives commands in the form of a series of 16-bit command words, known as a command packet. Each packet starts with a header, which is followed with any operation-specific data. The instrument operation selected by the command is specified using a Command Opcode field in the header.

The command manager assumes that all command packets appear as a series of 16-bit words, starting with the following header:

TABLE 19. Command Packet Header

Word	Field	Description	Min. Value	Max. Value
0	Packet Length	This field contains the length of the packet in 16-bit words.	3	256
1	Packet Identifier	This field contains an arbitrary number which the instrument uses solely for logging purposes.	0	65,535
2	Command Opcode	This field identifies which command to execute.	0	63
3 - (Packet Length - 1)	Data	The remainder of the packet contains opcode-specific data.		

This format is encapsulated using the **`CmdPkt`** class. The **`CmdPkt`** class provides access functions to the length (`getPacketLength`), identifier (`getPacketId`) and opcode fields (`getOpcode`), and also provides functions which return a pointer to the data area (`getDataAddress`), and the length of the contained data (`getDataLength`). In order to support echoing of commands, the **`CmdPkt`** class also provides access functions which return a pointer to the raw command buffer (`getBufferAddress`). NOTE: The length of the packet within the buffer can be determined using `getPacketLength`.

16.4.2 Processing Time

In order to eliminate the need to have interrupt handlers drain the command FIFO, the instrument software assumes that commands will not be sent faster than one command packet per Minor Telemetry Frame (i.e. about 4 commands per second), and will accept the real-time requirement that the **`CmdManager`** class must be capable of reading a command from the **`CmdDevice`** within 1 Minor Telemetry Frame (~250ms) of being written into the FIFO (including all interrupt overhead).

NOTE: The baseline requirement is to ensure that all commands be disposed of by the **`CmdManager`** in under 250ms. If a certain command causes the **`CmdManager`** to miss this deadline, then the additional delay required by the command shall be published in the ACIS Software Operators Manual.

16.4.3 Delays between groups of packets

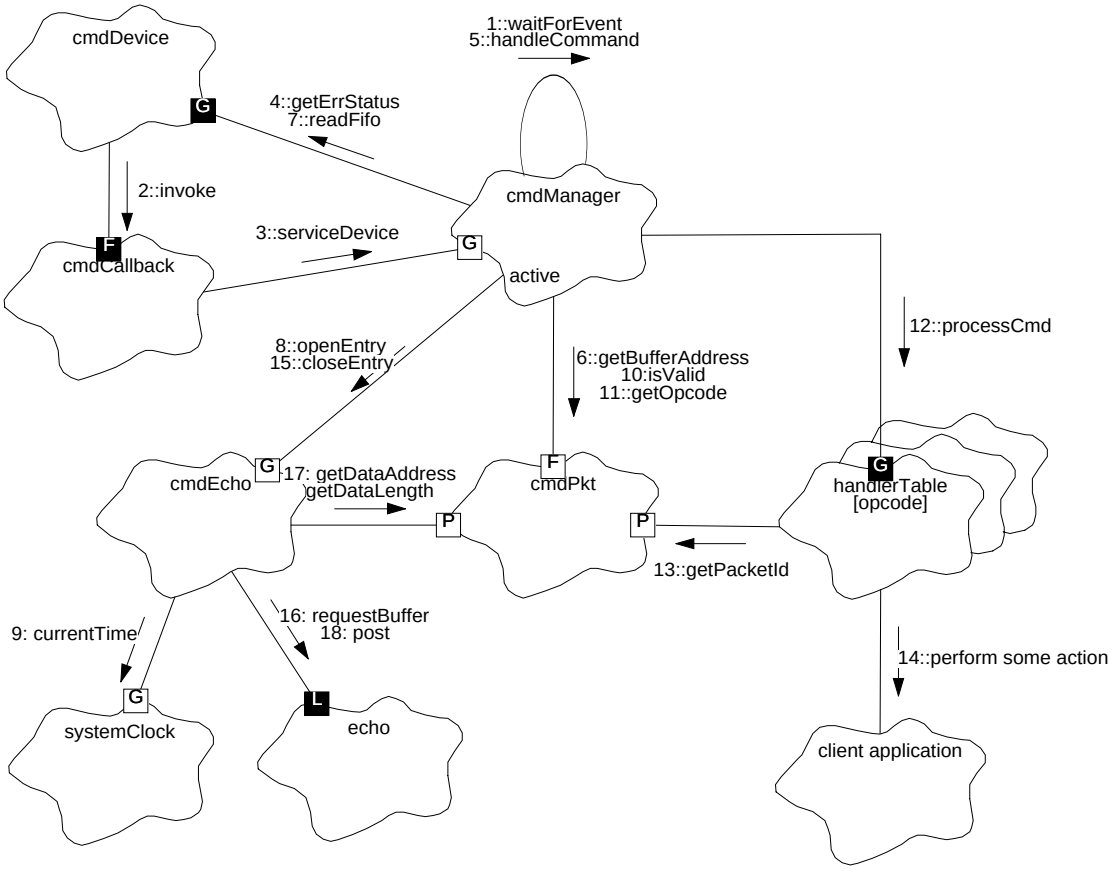
The ACIS instrument software uses a time-based method of grouping sets of related command packets. If an error occurs while receiving a particular command, this delay is used to determine the start of the next unrelated command packet. After an error is detected, all packets which arrive within this time limit from one another will be discarded. This delay time is approximately 1 second.

16.5 Scenarios

16.5.1 Use 1::Read, execute, and echo commands

Figure 61 illustrates the overall command processing scenario.

FIGURE 61. Command Processing Scenario



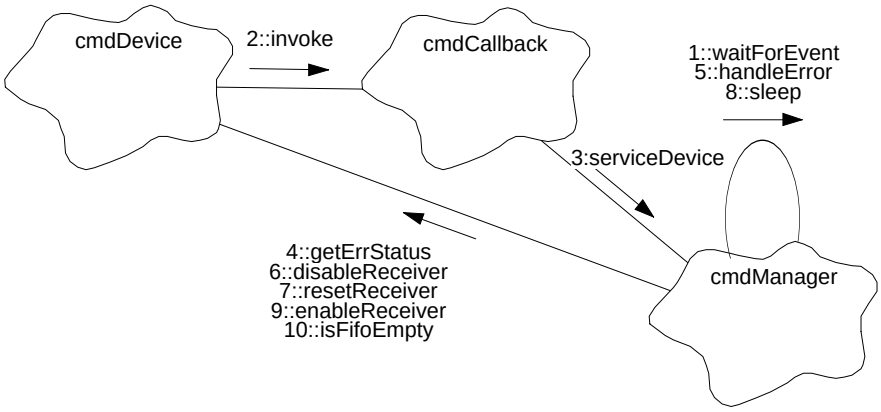
1. The *cmdManager* starts up and proceeds to wait for an event indicating activity from the *cmdDevice* (inherited **Task**::waitForEvent()).
2. Meanwhile, a command packet is acquired by the hardware and generates a command interrupt. The *cmdDevice* handles the interrupt and, as part of its interrupt processing, invokes the installed *cmdCallback*'s invoke() member function.
3. *cmdCallback* then invokes *cmdManager*.serviceDevice(). serviceDevice() then counts the interrupt and uses notify() to wake up the *cmd-Manager* task (not shown). The *cmdManager*'s is then woken up (appearing as a return from waitForEvent()).
4. The *cmdManager* then tests for any errors on the *cmdDevice* using **CmdDevice**::getErrStatus(). If an error is detected, the *cmdManager* invokes its recovery routine, handleError() (not shown). For the purposes of this scenario, assume that the device reported no errors.

5. If no errors are present, the *cmdManager* invokes its handleCommand() function to read and execute the command.
6. handleCommand() then queries a local **CmdPkt** for its packet buffer address, using **CmdPkt**::getBufferAddress().
7. handleCommand() then reads the packet length and data from the *cmdDevice* into the acquired packet buffer using **CmdDevice**::readFifo().
8. The *cmdManager* then asks the *cmdEcho* to record the arrival time of the packet, using **CmdEcho**::openEntry().
9. *cmdEcho*.openEntry() uses *systemClock*.currentTime() to obtain the current BEP timer tick and records the returned value.
10. The *cmdManager* then asks the now filled *cmdPkt* to perform a simple sanity check on its contents, using **CmdPkt**::isValid(). If the packet is invalid, the *cmdManager* drops into its command interface error handling code, handleError() (not shown). Assume for the purposes of this scenario, that the read packet data is valid.
11. The *cmdManager* queries the packet for its opcode using **CmdPkt**::getOpcode().
12. The *cmdManager* then uses the read opcode as index into a table of pointers to command handlers. It then invokes the handler's processCmd() member function, passing a pointer to the *cmdPkt* packet.
13. The handler then casts the buffer to the appropriate command packet class, and obtains the packet identifier (getPacketId) and other command-specific information from the instance.
14. The handler then performs command-specific operations, some of which may invoke member functions of other classes within the system.
15. Once the handler returns, the *cmdManager* passes the returned code to *cmdEcho*.closeEntry() to close out the entry for the command and record the disposition of the command.
16. *cmdEcho*.closeEntry() declares an echo telemetry packet builder, echo, and request a telemetry packet using *echo*.requestBuffer(). If it fails to obtain a buffer, the occurence is reported to the software housekeeper (not shown).
17. If a buffer is obtained, *cmdEcho*.closeEntry() retrieves command header information (not shown), and the command's data buffer address and length using *cmdPkt*.getDataAddress() and *cmdPkt*.getDataLength(). It then copies the information into the telemetry packet buffer.
18. Once the echo has been built. *cmdEcho*.closeEntry() posts the packet to telemetry using *echo*.post().

16.5.2 Use 2:: Recover from command errors

In order to locate the start of a command after detecting a command device or header format error, the **CmdManager** class uses an approach which relies on a minimum time-delay between decoupled command packets. Figure 62 illustrates this approach.

FIGURE 62. Command Recovery Scenario



1. The *cmdManager* is in its main loop, waiting for something to happen, using the inherited function, **Task**::waitForEvent().
2. An error occurs on the *cmdDevice* causing an interrupt. Its interrupt handler then invokes the installed callback *cmdCallback*.invoke().
3. The callback then invokes *cmdManager*.serviceDevice(), which then wakes up the *cmdManager* task using *cmdManager*.notify() (not shown). The *cmdManager* then returns from its waitForEvent() call.
4. The *cmdManager* tests the *cmdDevice* for an error, using *cmdDevice*.getErrStatus(). In this scenario, an error is present.
5. The *cmdManager*, upon detecting an error from the *cmdDevice*, invokes its handleError() function.
6. handleError() then enters a loop, where it disables the *cmdDevice* interrupt generation logic using **CmdDevice**::disableReceiver(). NOTE: Command words received by the instrument while the receiver is disabled are still written into the FIFO, but will not cause a command interrupt).
7. handleError() resets the error condition and clears the FIFO using **CmdDevice**.resetReceiver()
8. The error loop then sleeps (**Task**::sleep()) for a period of time, determined by the **CmdManager** variable, *pktDelay* (not shown).
9. handleError() then re-enables the receiver using **CmdDevice**::enableReceiver().

10. `handleError`() then tests the command FIFO using *`cmdDevice`*`.isFifoEmtpy`(). If the FIFO is not empty, or if the error condition persists (*`cmdDevice`*`.getErrStatus`()), `handleError`() repeats the process from the point of disabling the receiver. If the FIFO is empty, and the error condition is cleared, `handleError`() returns and the *`cmdManager`* proceeds to wait for the next command. NOTE: All software housekeeping reports in this scenario are TBD.

16.6 Class CmdManager

Documentation:

The Command Manager is responsible for reading commands from the Command Device and for executing the received commands.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Implementation Uses:

`CmdDevice`
`CmdPkt`
`CmdEcho`

Public Interface:

Operations:

`CmdManager()`
`goTaskEntry()`
`setHandler()`
`serviceDevice()`

Constants:

RECOVER_SECONDS=1 second

Protected Interface:

Has-A Relationships:

`CmdHandler` *`handlerTable`*[]: This table is an array of pointers to Command Handler instances. The array is indexed by command opcode.

`unsigned` *`pktPending`*: This is the number of command packets contained in the command FIFO. It is incremented with each call to serviceDevice(), is decremented when each packet is processed, and is set to zero during error processing.

`const unsigned` *`pktDelay`*: This read-only variable contains the number of clock ticks (1/10 second) to wait after an error to ensure the start of a subsequent packet. This value is set to RECOVER_SECONDS converted into clock ticks (i.e. multiplied by 10).

Operations: `handleCommand()`
`handleError()`

Concurrency: Active

Persistence: Persistent

16.6.1 CmdManager()

Public member of: **`CmdManager`**

Arguments:

`unsigned` *`taskId`*

Documentation:

This constructor initializes the task information. *`taskId`* is the ***`Nucleus RTX`*** task identifier for the **`CmdManager`**.

Semantics:

Invoke the parent's **`Task`**::`Task`() constructor, passing *`taskId`* as the argument and initialize the read-only *`pktDelay`* variable to 1 second (10 timer ticks), and zero *`pktPending`*.

Concurrency: Sequential

16.6.2 goTaskEntry()

Public member of: **`CmdManager`**

Return Class: **`void`**

Documentation:

This function contains the main loop of the Command Manager task. Its loop must iterate at least once per 250ms in order to never miss a command. This function NEVER returns.

Semantics:

Within a FOREVER loop, wait for an event (using **`Task`**::`waitForEvent`()). If a task monitor query is present, respond to the query. If the command device wants attention, test for errors using *`cmdDevice`*.`getErrStatus`(). If an error is present, call `handleError`(), otherwise, call `handleCommand`() to process any pending commands

Time complexity: When commands are present, must be able to iterate in under 250ms

Concurrency: Synchronous

16.6.3 handleCommand()

Protected member of: **`CmdManager`**

Return Class: **`void`**

Documentation:

This function reads and executes one command from the Command Device.

Semantics:

Read the packet length from the command FIFO (*`cmdDevice`*`.readFifo`()). If length invalid, call `handleError`() and return. If length valid, get packet buffer address, store length and read remainder of packet. Test the command header for legal values (*`cmdDevice`*`.isValid`()). If header invalid, call `handleError`() and return. If header ok, open the command echo entry (*`cmdEcho`*`.openEntry`()). Get the packet opcode and index the handler table. If the table entry is 0, log command un-implemented. If not 0, invoke the handler (*`handler`*`->processCmd`()), and echo the command and the returned result (*`cmdEcho`*`.closeEntry`()). Decrement *`pktPending`*. If *`pktPending`* is not zero, notify the task of another command using `notify`().

Time complexity: Must execute in under 250ms

Concurrency: Synchronous

16.6.4 handleError()

Protected member of: **`CmdManager`**

Return Class: **`void`**

Documentation:

This function attempts to recover from errors in the Command Device.

Semantics:

Do the following until there are no command errors reported and until the FIFO remains empty: disable the receive logic and reset the FIFO, sleep for the packet delay time, re-enable the receive logic. Zero *`pktPending`*.

Concurrency: Synchronous

16.6.5 serviceDevice()

Public member of: **`CmdManager`**

Return Class: **`void`**

Documentation:

This function is responsible for notifying the task portion of the CmdManager when a command packet arrives. When it is invoked, it increments *`pktPending`*. It then uses `notify`() to issue an event to the task.

Concurrency: Synchronous

16.6.6 setHandler()

Public member of: **CmdManager**

Return Class: **void**

Arguments:

CmdOpcode *opcode*
CmdHandler* *handler*

Documentation:

This function writes the *handler* pointer into the *handlerTable* slot indexed by *opcode*, overwriting any previous handler pointer occupying that slot. If *handler* is 0, then commands corresponding to *opcode* are rejected by the *cmdManager*.

Concurrency: Sequential

16.7 Class CmdPkt

Documentation:

Command Packets are used to instruct the ACIS software to perform some action. The top-level command packet acts as the transport layer into the ACIS software.

All Command Packets have the following, overall format:

Word 0: Length
Word 1: Packet Identifier
Word 2: Command Opcode
Words[Length-3]: Command opcode-specific data

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:

```
getBufferAddress()
getDataAddress()
getDataLength()
getOpcode()
getPacketId()
getPacketLength()
isValid()
```

Private Interface:

Has-A Relationships:

`unsigned short` *`pktBuffer`*[256]: This buffer contains the list of 16-bit words from the command packet.

Concurrency: Guarded

Persistence: Transient

16.7.1 getBufferAddress()

Public member of: **CmdPkt**

Return Class: **unsigned short***

Documentation:

This function returns a pointer to the packet buffer. This buffer is at least 256 16-bit words in length.

Concurrency: Guarded

16.7.2 getDataAddress()

Public member of: **CmdPkt**

Return Class: **const unsigned short***

Documentation:

If the packet contains data, this function returns a read-only pointer to the command packet's data area. The length of this area is determined by calling `getDataLength`(). If the packet's length does not support any data, this function returns 0.

Concurrency: Guarded

16.7.3 getDataLength()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

This function returns the number of 16-bit words in the packet's data area. The address of this area is obtained using `getDataAddress`().

Concurrency: Guarded

16.7.4 getOpcode()

Public member of: **CmdPkt**

Return Class: **enum CmdOpcode**

Documentation:

If the packet length is valid, this function returns the opcode contained within the command packet, otherwise it returns CMDOP_INVALID.

Concurrency: Guarded

16.7.5 getPacketId()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

If the packet's length is valid, this function returns the packet identifier contained within the command packet, otherwise it returns 0xffffffff.

Concurrency: Guarded

16.7.6 getPacketLength()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

This function returns the total number of 16-bit words delivered by the command packet. The address of the raw packet buffer is provided by `getBufferAddress`().

Concurrency: Guarded

16.7.7 isValid()

Public member of: **`CmdPkt`**

Return Class: **`Boolean`**

Documentation:

This function performs a top level sanity check on its contents. It returns:
`BoolTrue` - Sanity check passed
`BoolFalse` - Sanity check failed

Concurrency: Guarded

16.8 Class CmdHandler

Documentation:

A command handler is responsible for handling a given command from the ground. This class is an abstract class, and is intended to serve as a top level command handler template. All command handlers must be a subclass of this type.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

CmdPkt
CmdEcho

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

16.8.1 processCmd()

Public member of: **CmdHandler**

Return Class: **enum CmdResult**

Arguments:

CmdPkt* *pkt*

Documentation:

This is an abstract virtual function. Subclasses of **CmdHandler** must implement this function to execute the command indicated by the referenced command packet, and must return the appropriate **CmdResult** code indicating the disposition of the command.

Time complexity: All `processCmd`() implementations must execute in under 250ms

Concurrency: Synchronous

16.9 Class CmdEcho

Documentation:

The Command Echo is responsible for echoing a command packet to telemetry.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Uses:

CmdPkt

Implementation Uses:

SystemClock
Tf_Command_Echo

Public Interface:

Operations: `closeEntry()`
`openEntry()`

Private Interface:

Has-A Relationships:

CmdPkt* *curpkt*: This points to the command packet registered via `openEntry`().

unsigned *arrivalTime*: This indicates the approximate arrival time of the command, in BEP timer ticks.

Concurrency: Guarded

Persistence: Persistent

16.9.1 closeEntry()

Public member of: **CmdEcho**

Return Class: **void**

Arguments:

CmdPkt* *pkt*
CmdResult *disposition*

Documentation:

This function forms and issues an echo of the command packet. The packet is indicated by *pkt*, and the exeuction disposition is indicated by *disposition*.

Preconditions:

pkt must match *curpkt* and must not be zero

Semantics:

Declare a command echo telemetry builder. Request a telemetry buffer for the instance. If successful, store the *arrivalTime* and *disposition* into the buffer, get the command header info from *pkt* and store into the buffer, and then copy the command body into the telemetry buffer. Once built, post the buffer to telemetry. If no buffer is available, report to software housekeeping and skip the echo.

Concurrency: Guarded

16.9.2 openEntry()

Public member of: **CmdEcho**

Return Class: **void**

Arguments:

CmdPkt* *pkt*

Documentation:

This function logs the arrival time of the passed command packet, *pkt*. It copies *pkt* into its private *curpkt*. and then uses *systemClock*.currentTime() to obtain the current BEP timer tick. It then stores the timer tick count into *arrivalTime*.

Concurrency: Guarded

17.0 Command Packet Handler Classes (36-53214 A)

17.1 Purpose

The purpose of the command handler classes are to interpret the contents of received command packets and execute the operations mandated by the command packet operation code. This section describes the command handler class design. The details of the command packet formats are shown in the “ACIS Instrument Program and Command List,” MIT 36-01410 (IP&CL).

17.2 Uses

Use 1:: Execute Parameter Block Load Commands
Use 2:: Initiate Memory Server Actions
Use 3:: Initiate DEA Housekeeping and Science Actions
Use 4:: Execute Patch Commands
Use 5:: Execute System Configuration Commands

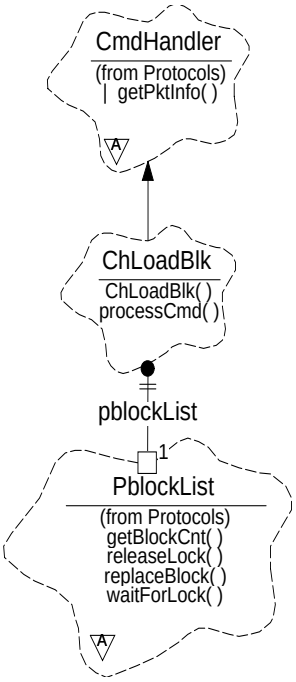
The commands used during boot to load an image from uplink and execute the loaded image are not described in this section. Refer to the IP&CL for a description of these particular command formats and their scenarios. NOTE: Command handlers cannot assume that unused bits within command packet words are zero, and shall mask off bits within words that they do not use.

17.3 Organization

All command handlers are a subclass of ***`Protocols`***::**`CmdHandler`** (see Section 16.0). Each subclass of **`CmdHandler`** implements its own `processCmd`() member function. Given the large number of command handler classes, the following figures illustrate the class relationships involved in each of the use-cases listed above.

17.3.1 Parameter Block Command Handler

FIGURE 63. Parameter Block Command Handler Classes

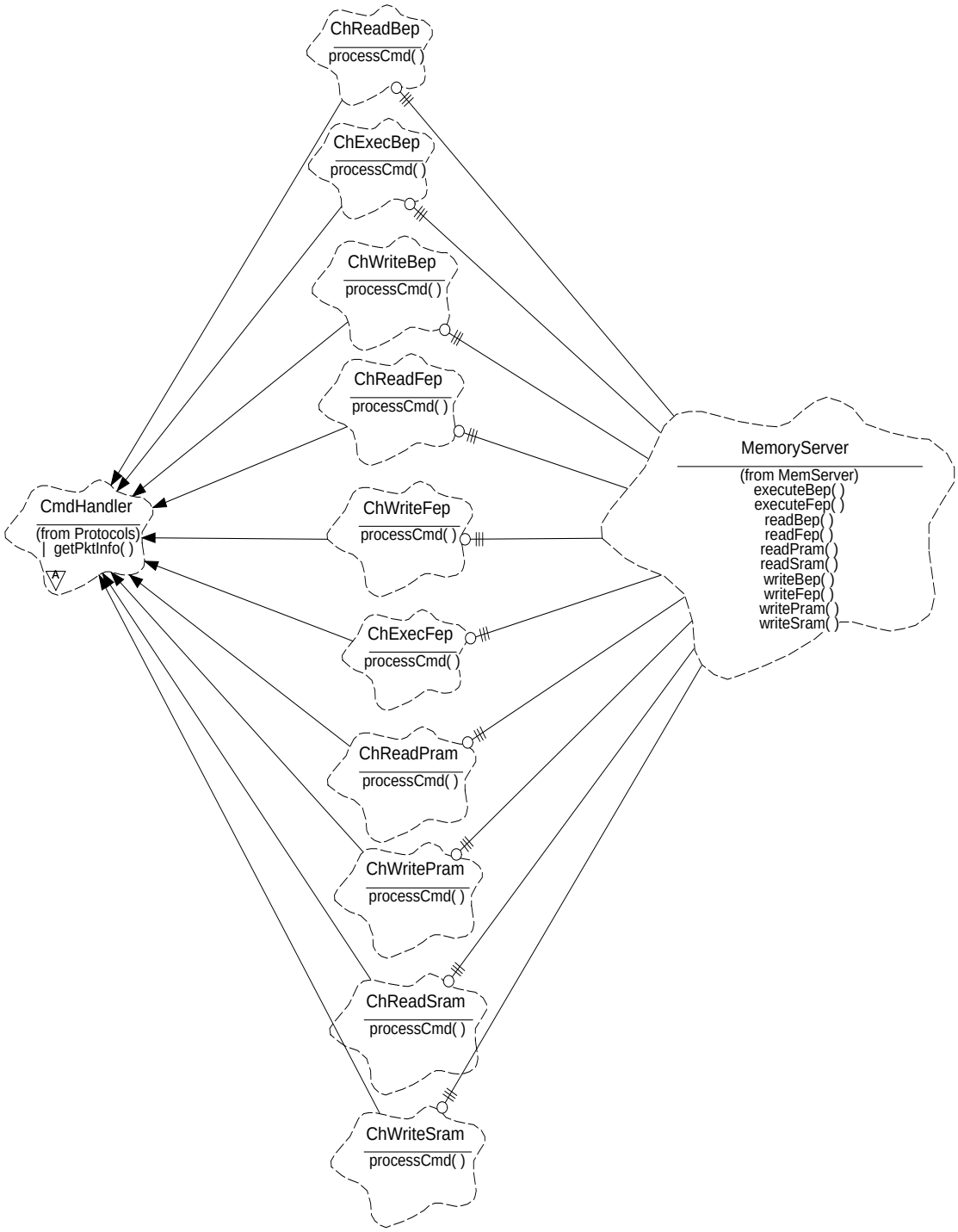


`ChLoadBlk`- This class is a subclass of **`CmdHandler`** and represents all command handlers which load either Science or DEA parameter blocks into the instrument. The constructor for this handler class requires a command opcode and a reference to a parameter block list (**`PblockList`**). The constructed instance associated with the opcode is then responsible for loading parameter blocks from incoming commands into its parameter block list. This class uses the **`PblockList`** class to store the loaded parameter block. The **`ChLoadBlk`** class has knowledge about the common header format all parameter blocks, but leaves the interpretation of the body of the parameter block up to the specific parameter block type maintained by the targeted parameter block list.

`PblockList`- This class is responsible for providing the top-level interface to all types of parameter blocks within the instrument. The top level functions provided by this class are used by the command handler class to install parameter blocks, and by Science and DEA Housekeeping to retrieve loaded parameter blocks. This class is described in detail in Section TBD.

17.3.2 Memory Server Command Handlers

FIGURE 64. Memory Server Command Handler Classes



`ChReadBep`- This is a subclass of **`CmdHandler`** and is responsible for handling commands to read and telemeter data and/or code from the Back End Processor's RAM. This class uses the **`MemoryServer`** class to perform the actual operation.

ChWriteBep- This is a subclass of **CmdHandler** and is responsible for handling commands to write data (and possibly code) to the Back End Processor's RAM. This class uses the **MemoryServer** class to perform the actual operation.

ChExecBep- This is a subclass of **CmdHandler** and is responsible for handling commands instructing the instrument to call a subroutine located within the Back End Processor's RAM. This class uses the **MemoryServer** class to perform the actual operation.

ChReadFep- This is a subclass of **CmdHandler** and is responsible for handling commands to read and telemeter data (and possibly code) from one of the Front End Processors' RAM. This class uses the **MemoryServer** class to perform the actual operation.

ChWriteFep- This is a subclass of **CmdHandler** and is responsible for handling commands to write data (and possibly code) to one of the Front End Processors' RAM. This class uses the **MemoryServer** class to perform the actual operation.

ChExecFep- This is a subclass of **CmdHandler** and is responsible for handling commands instructing the instrument to call a subroutine located within one of the Front End Processors' RAM. This class uses the **MemoryServer** class to perform the actual operation.

ChReadPram- This is a subclass of **CmdHandler** and is responsible for handling commands to read and telemeter data from one of the Detector Electronics Assembly (DEA) CCD Controller's Program RAM (PRAM). This class uses the **MemoryServer** class to perform the actual operation.

ChWritePram- This is a subclass of **CmdHandler** and is responsible for handling commands to write data to one of the DEA CCD Controller's Program RAM. This class uses the **MemoryServer** class to perform the actual operation.

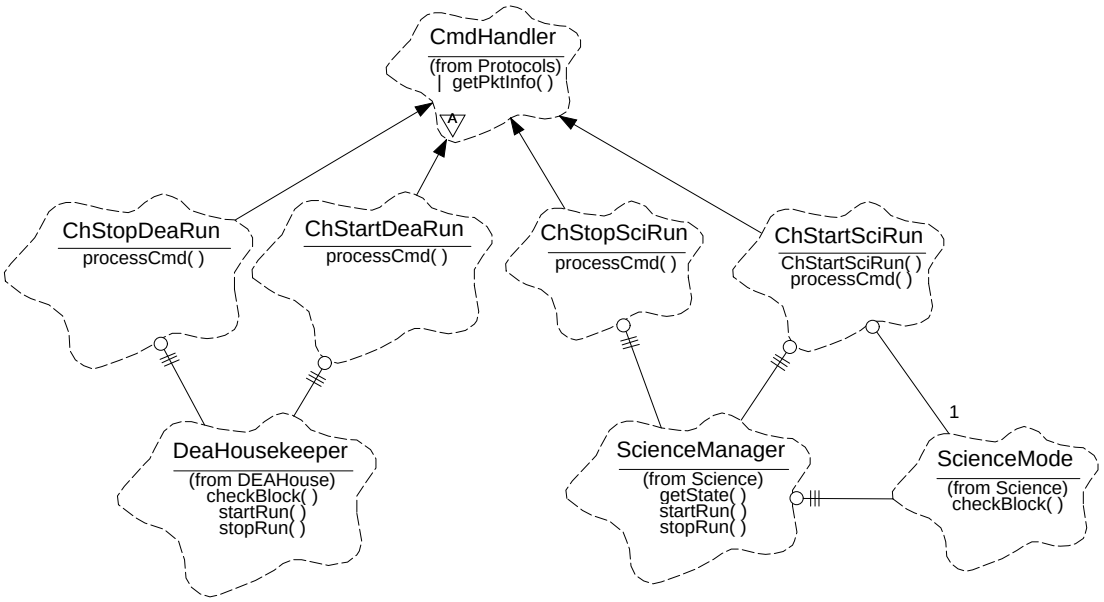
ChReadSram- This is a subclass of **CmdHandler** and is responsible for handling commands to read and telemeter data from one of the DEA CCD Controller's Sequencer RAM (SRAM). This class uses the **MemoryServer** class to perform the actual operation.

ChWriteSram- This is a subclass of **CmdHandler** and is responsible for handling commands to write data to one of the DEA CCD Controller's Sequencer RAM. This class uses the **MemoryServer** class to perform the actual operation.

MemoryServer- This class is a subclass of ***Executive***::**Task** (not shown), and is responsible for performing or initiating all memory loads, dumps and executes within ACIS. This class is described in detail in Section 27.0.

17.3.3 Science Run and DEA Housekeeping Command Handlers

FIGURE 65. Science Run and DEA Housekeeping Command Handler Classes



ChStartSciRun- This class is a subclass of **CmdHandler**, and is responsible for handling commands initiating a science run. Upon construction, each instance of this class is passed a command opcode, a reference to a **ScienceMode** instance, and a flag indicating whether or not the command requests only a bias computation, with no subsequent data processing. Once a command is received, the handler instance associated with the command's opcode uses installed **ScienceMode** and bias-only flag to instruct the **ScienceManager** what type of science run was requested.

ChStopSciRun- This class is a subclass of **CmdHandler**, and is responsible for terminating an active science run. There is one instance of this class for each type of science run. This class uses the **ScienceManager** to terminate the run. NOTE: If a science command is running, but the stop request is of the wrong type or specifies the wrong parameter block slot id, the current run will be stopped anyway, and a software housekeeping error statistic will be reported. When a run is stopped, it finishes its current exposure. The run's telemetry packets remain in the telemetry queue until they are sent out of the instrument by the telemetry manager.

ChStartDeaRun- This class is a subclass of **CmdHandler**, and is responsible for handling commands that initiate acquisition of DEA Housekeeping data. This class uses the **DeaHousekeeper** to perform the operation.

ChStopDeaRun- This class is a subclass of **CmdHandler**, and is responsible for handling commands that request the termination of DEA housekeeping acquisition. This class uses the **DeaHousekeeper** class to terminate the operation. NOTE: If the stop request specifies the wrong parameter block slot id, the current run will be stopped anyway, and a software housekeeping error statistic will be reported. When a run is stopped, it finishes its

current housekeeping packet. The run’s telemetry packets remain in the telemetry queue until they are sent out of the instrument by the telemetry manager.

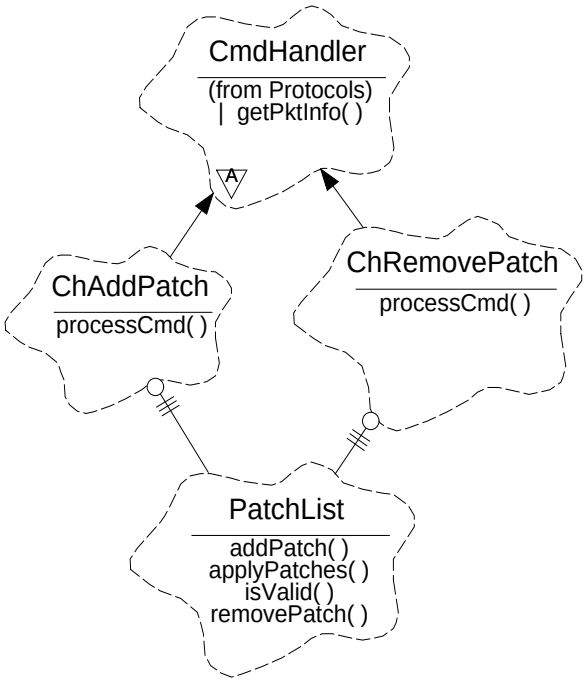
`ScienceManager`- This class is a subclass of ***`Executive`***::**`Task`** (not shown).This class is responsible for coordinating science bias computations and data runs. See Section 33.0 for a more detailed description of this class.

`ScienceMode`- This class is responsible for implementing the detailed setup, parameter dump, bias computation, data processing, and cleanup operations of a particular science mode. This class operates under the direction of the **`ScienceManager`** class. This class is described in more detail in Section 33.0.

`DeaHousekeeper`- This class is a subclass of ***`Executive`***::**`Task`** (not shown), and is responsible for periodically acquiring and telemetering DEA housekeeping information. This class is described in more detail in Section TBD.

17.3.4 Patch Command Handlers

FIGURE 66. Patch Command Handler Classes



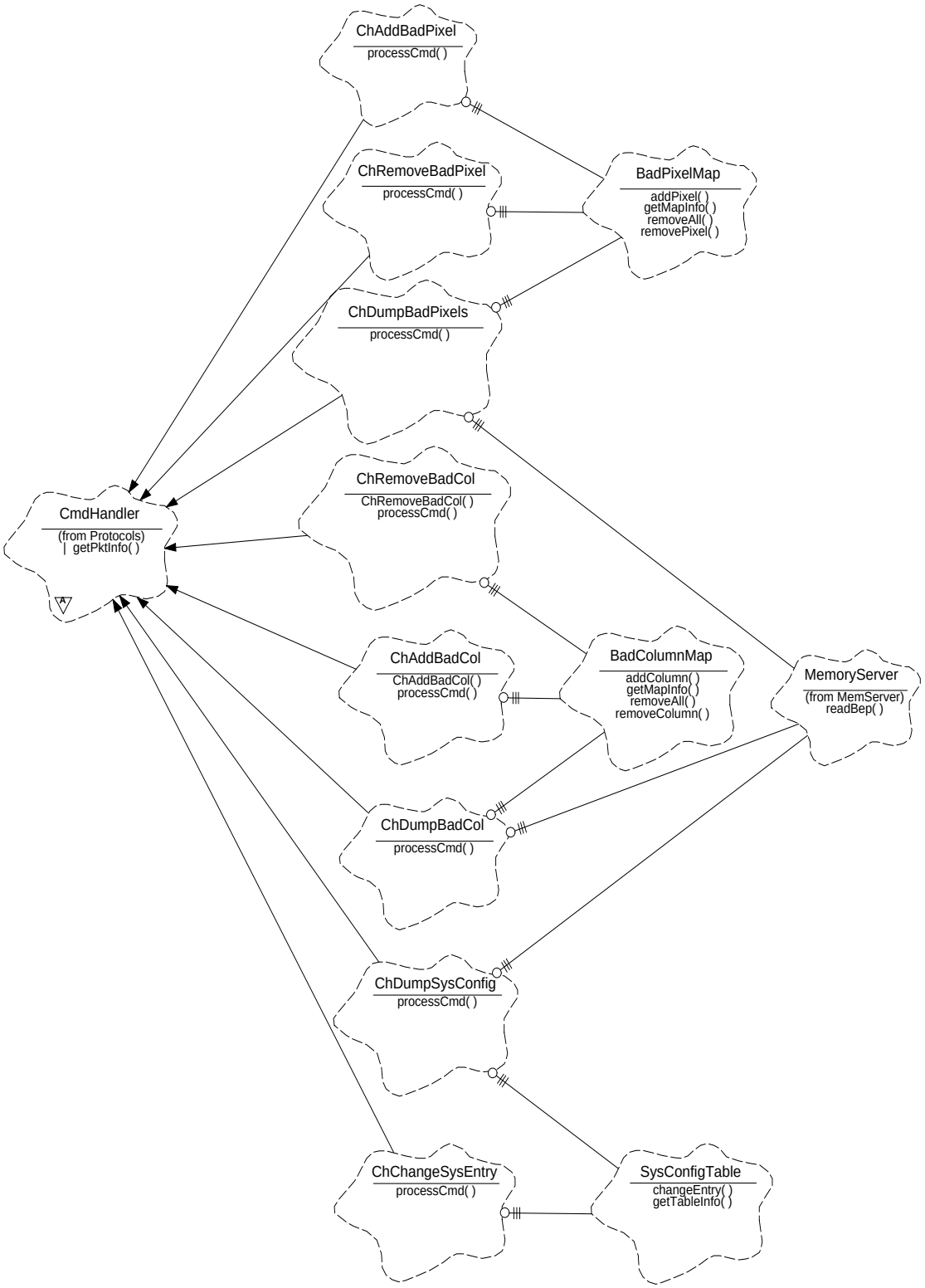
`ChAddPatch`- This class is a subclass of **`CmdHandler`** and is responsible for handling commands which install new patches into the ACIS patch list. This class uses the **`PatchList`** class to add the requested patch to the system. The installed patch will not take effect until the next commanded reset.

`ChRemovePatch`- This class is a subclass of **`CmdHandler`** and is responsible for removing a patch from the ACIS software patch list. This class uses the **`PatchList`** class to remove the patch from the system. The code or data affected by the removed patch will not be restored until the next commanded reset.

`PatchList`- This class is responsible for maintaining the ACIS software patch list. It is described in more detail in Section TBD.

17.3.5 System Configuration Command Handlers

FIGURE 67. System Configuration Command Handler Classes



Within ACIS, there is one bad pixel map, for use with Timed Exposure Mode, and two bad column maps, one for Timed Exposure Mode, and one for use with Continuous Clocking Mode.

ChAddBadPixel- This class is a subclass of **CmdHandler** and is responsible for handling commands which install one or more bad pixels in the system's bad pixel list. This class uses the **BadPixelMap** class to maintain the current list of bad pixels.

ChRemoveBadPixel- This class is a subclass of **CmdHandler**, and is responsible for handling commands which remove pixels from the system's bad pixel list, or which clear the entire list. This class uses the **BadPixelMap** class to maintain the current list of bad pixels.

ChDumpBadPixels- This class is a subclass of **CmdHandler**, and is responsible for dumping the contents of the bad pixel map to telemetry. This class uses the **BadPixelMap** class to obtain access to the address and current length of the bad pixel map buffer. This class then uses the **MemoryServer** class to iteratively format and send memory dump packets of the bad pixel map.

ChAddBadCol- This class is a subclass of **CmdHandler**, and is responsible for adding one or more bad columns to a bad column map. Upon construction, each instance of this handler is associated with a command opcode and a **BadColumnMap** instance. Each handler instance uses its associated **BadColumnMap** instance to maintain the list of bad columns.

ChRemoveBadCol- This class is a subclass of **CmdHandler**, and is responsible for removing one or more bad columns from a bad column map, or clear all bad columns from the corresponding map. Upon construction, each instance of this handler is associated with a command opcode and a **BadColumnMap** instance. Each handler instance uses its associated **BadColumnMap** instance to maintain the list of bad columns.

ChDumpBadCol- This class is a subclass of **CmdHandler**, and is responsible for dumping the contents of the bad pixel map to telemetry. Upon construction, each instance of this handler is associated with a command opcode and a **BadColumnMap** instance. Each handler instance uses its associated **BadColumnMap** instance to obtain the address and current length of a given bad column map. The handler uses the **MemoryServer** class to iteratively format and dump the contents of the map to telemetry.

ChChangeSysEntry- This class is a subclass of **CmdHandler**, and is responsible for handling commands which change one or more values of the System Configuration Table. This class uses the **SysConfigTable** class to maintain the current system settings.

ChDumpSysConfig- This class is a subclass of **CmdHandler**, and is responsible for handling commands which request a dump of the current System Configuration Table entries. The handler class uses the **SysConfigTable** class to obtain the address and length of the table, and uses the **MemoryServer** to format and dump the table to telemetry.

17.4 Command Packet Handling Restrictions

17.4.1 Handling Restrictions

All command packet header information is accessed via the **`CmdPkt`** class (see Section 16.0). Unless otherwise specified, all command handler `processCmd`() member functions are required to execute in under 200ms. If a command takes longer than 200ms to execute, the handler should consider forwarding the request onto a lower priority task to execute.

In order to make interpretation of command packets easier, all packets must align long word data elements on long word boundaries. All command packet buffers will start on a long word boundary, and the command packet header will always be an odd number of 16-bit words in length.

See Section 4.6 for a detailed description of all command packet formats defined by ACIS.

17.4.2 Command Handler Object Map

The following table maps each command packet type to a specific object name and class.

TABLE 20. Command Packet to Handler Object Map

Command	Opcode Symbol	Object Name	Class Name
Load Timed Exposure Parameter Block	CMDOP_LOAD_TE	chLoadTimedExp	ChLoadBlk
Load Cont. Clocking Parameter Block	CMDOP_LOAD_CC	chLoadContClock	ChLoadBlk
Load 2-D Window List	CMDOP_LOAD_2D	chLoad2dWindow	ChLoadBlk
Load 1-D Window List	CMDOP_LOAD_1D	chLoad1dWindow	ChLoadBlk
Load DEA House.Parameter Block	CMDOP_LOAD_DEA	chLoadDeaHouse	ChLoadBlk
Read BEP Memory	CMDOP_READ_BEP	chReadBep	ChReadBep
Write BEP Memory	CMDOP_WRITE_BEP	chWriteBep	ChWriteBep
Execute BEP Memory	CMDOP_EXEC_BEP	chExecBep	ChExecBep
Read FEP Memory	CMDOP_READ_FEP	chReadFep	ChReadFep
Write FEP Memory	CMDOP_WRITE_FEP	chWriteFep	ChWriteFep
Execute FEP Memory	CMDOP_EXEC_FEP	chExecFep	ChExecFep
Read PRAM	CMDOP_READ_PRAM	chReadPram	ChReadPram
Write PRAM	CMDOP_WRITE_PRAM	chWritePram	ChWritePram
Read SRAM	CMDOP_READ_SRAM	chReadSram	ChReadSram
Write SRAM	CMDOP_WRITE_SRAM	chWriteSram	ChWriteSram
Start Timed Exposure	CMDOP_START_TE	chStartTimedExp	ChStartSciRun
Start Timed Exposure Bias Only	CMDOP_BIAS_TE	chBiasTimedExp	ChStartSciRun
Stop Timed Exposure	CMDOP_STOP_TE	chStopTimedExp	ChStopSciRun
Start Continuous Clocking	CMDOP_START_CC	chStartCc	ChStartSciRun
Start Continuous Clocking Bias Only	CMDOP_BIAS_CC	chBiasCc	ChStartSciRun
Stop Continuous Clocking	CMDOP_STOP_CC	chStopCc	ChStopSciRun

TABLE 20. Command Packet to Handler Object Map

Command	Opcode Symbol	Object Name	Class Name
Start DEA Housekeeping	CMDOP_START_DEA	chStartDeaRun	ChStartDeaRun
Stop DEA Housekeeping	CMDOP_STOP_DEA	chStopDeaRun	ChStopDeaRun
Add Patch	CMDOP_ADD_PATCH	chAddPatch	ChAddPatch
Remove Patch	CMDOP_REMOVE_PATCH	chRemovePatch	ChRemovePatch
Add Bad Pixel	CMDOP_ADD_BAD_PIXEL	chAddBadPixel	ChAddBadPixel
Remove Bad Pixel	CMDOP_REMOVE_BAD_PIXEL	chRemoveBadPixel	ChRemoveBadPixel
Dump Bad Pixels	CMDOP_DUMP_BAD_PIXELS	chDumpBadPixels	ChDumpBadPixels
Add Bad Timed Exposure Column	CMDOP_ADD_BAD_TE_COL	chAddBadTeCol	ChAddBadCol
Remove Bad Timed Exposure Column	CMDOP_REMOVE_BAD_TE_COL	chRemoveBadTeCol	ChRemoveBadCol
Dump Bad Timed Exposure Columns	CMDOP_DUMP_BAD_TE_COL	chDumpBadTeCol	ChDumpBadCol
Add Bad Continuous Clocking Column	CMDOP_ADD_BAD_CC_COL	chAddBadCcCol	ChAddBadCol
Remove Bad Cont. Clocking Column	CMDOP_REMOVE_BAD_CC_COL	chRemoveBadCcCol	ChRemoveBadCol
Dump Bad Cont. Clocking Columns	CMDOP_DUMP_BAD_CC_COL	chDumpBadCcCol	ChDumpBadCol
Change System Configuration Entry	CMDOP_CHANGE_SYS_ENTRY	chChangeSysEntry	ChChangeSysEntry
Dump System Configuration Entries	CMDOP_DUMP_SYS_CONFIG	chDumpSysConfig	ChDumpSysConfig

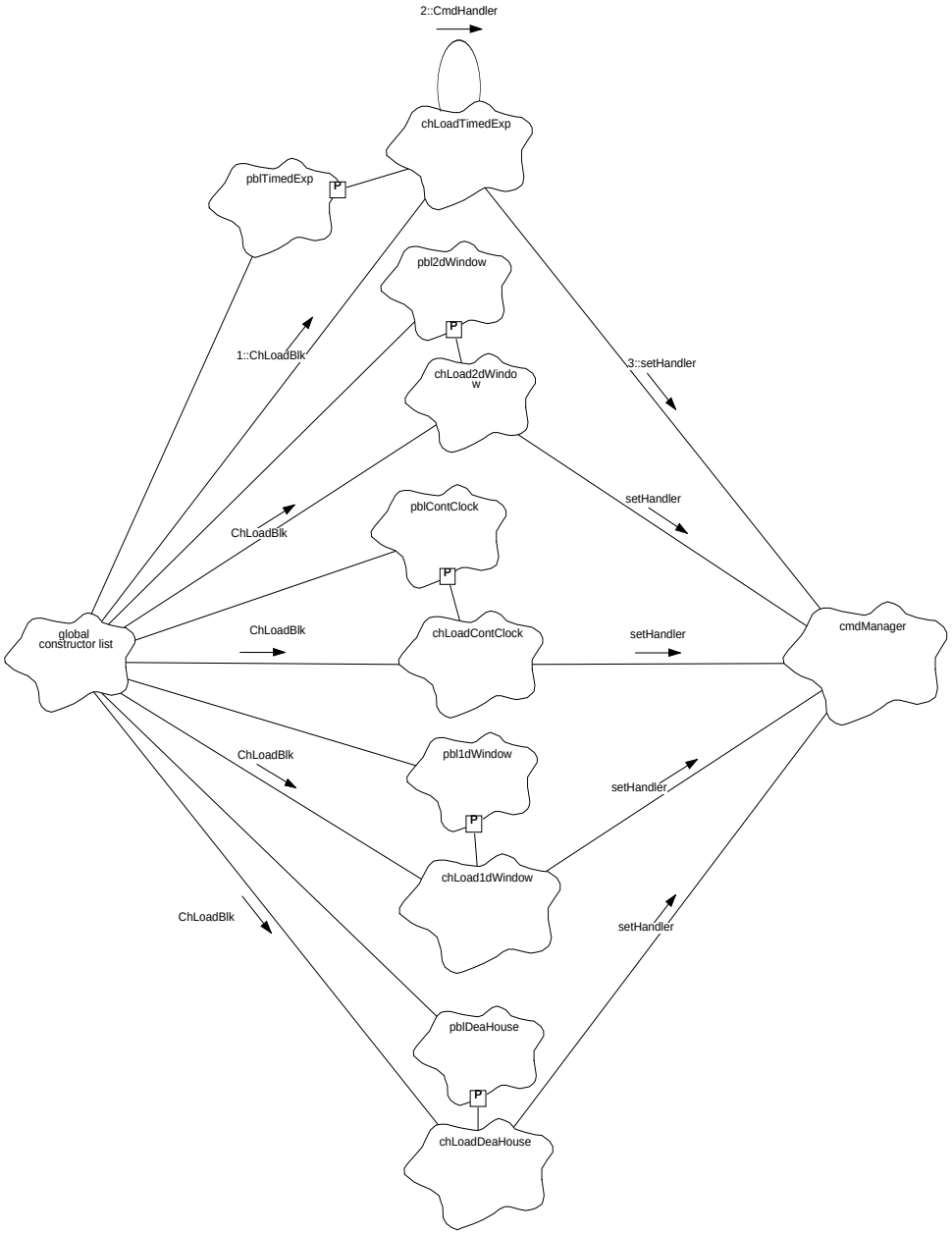
17.5 Scenarios

17.5.1 Use 1:: Load Parameter Blocks

17.5.1.1 Handler Object Construction

Each of the command handler objects are created during system initialization, and are registered with the Command Manager at that time. Figure 68 illustrates the overall initialization procedure used to construct command handlers which store parameter blocks within ACIS. The following scenario details the actions taken when constructing the “Load Timed Exposure Parameter Block” command handler. The scenario applies to all of the remaining constructors, by varying only the command operation codes and parameter block list references for each distinct command handler object.

FIGURE 68. Parameter Block Command Handler Initialization

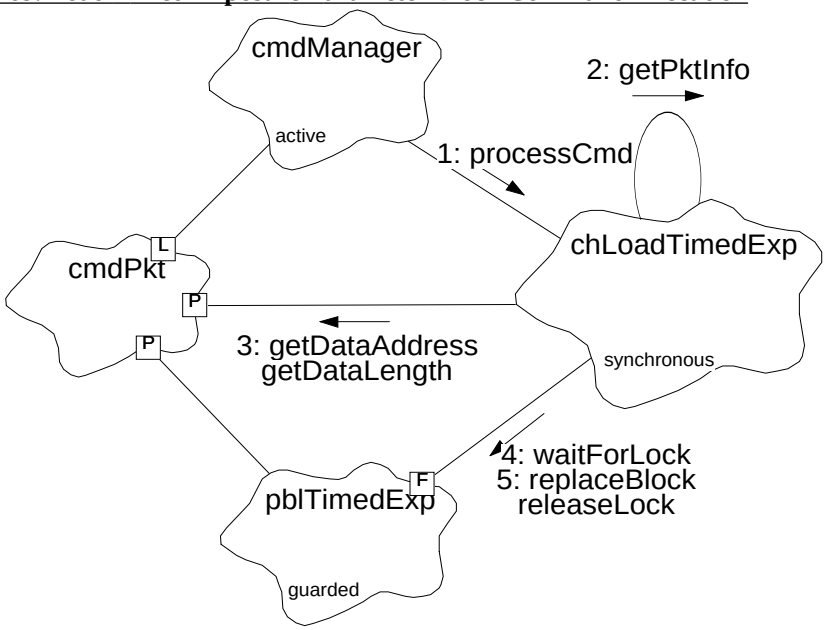


1. The global constructor list invokes the constructor for *chLoadTimedExp*, passing CMDOP_LOAD_TE as the command opcode, and a reference to *pblTimedExp* as the parameter block list affected by *chLoadTimedExp*.
2. *chLoadTimedExp*'s constructor then invokes its parent constructor, **CmdHandler**::CmdHandler, passing the supplied command operation code. Once its parent is initialized, *chLoadTimedExp* initializes its parameter block list pointer with the passed reference to *pblTimedExp*.
3. The **CmdHandler** constructor invokes the **CmdManager** function, setHandler(), to install itself as the handler for the supplied command operation code.

17.5.1.2 Execution of Load Parameter Block Commands

Figure 69 illustrates the overall execution of a “Load Timed Exposure Parameter Block Command.” Other types of “Load Parameter Block” commands operate in the same fashion, only using different command handler instances and destination parameter block lists.

FIGURE 69. Load Timed Exposure Parameter Block Command Execution

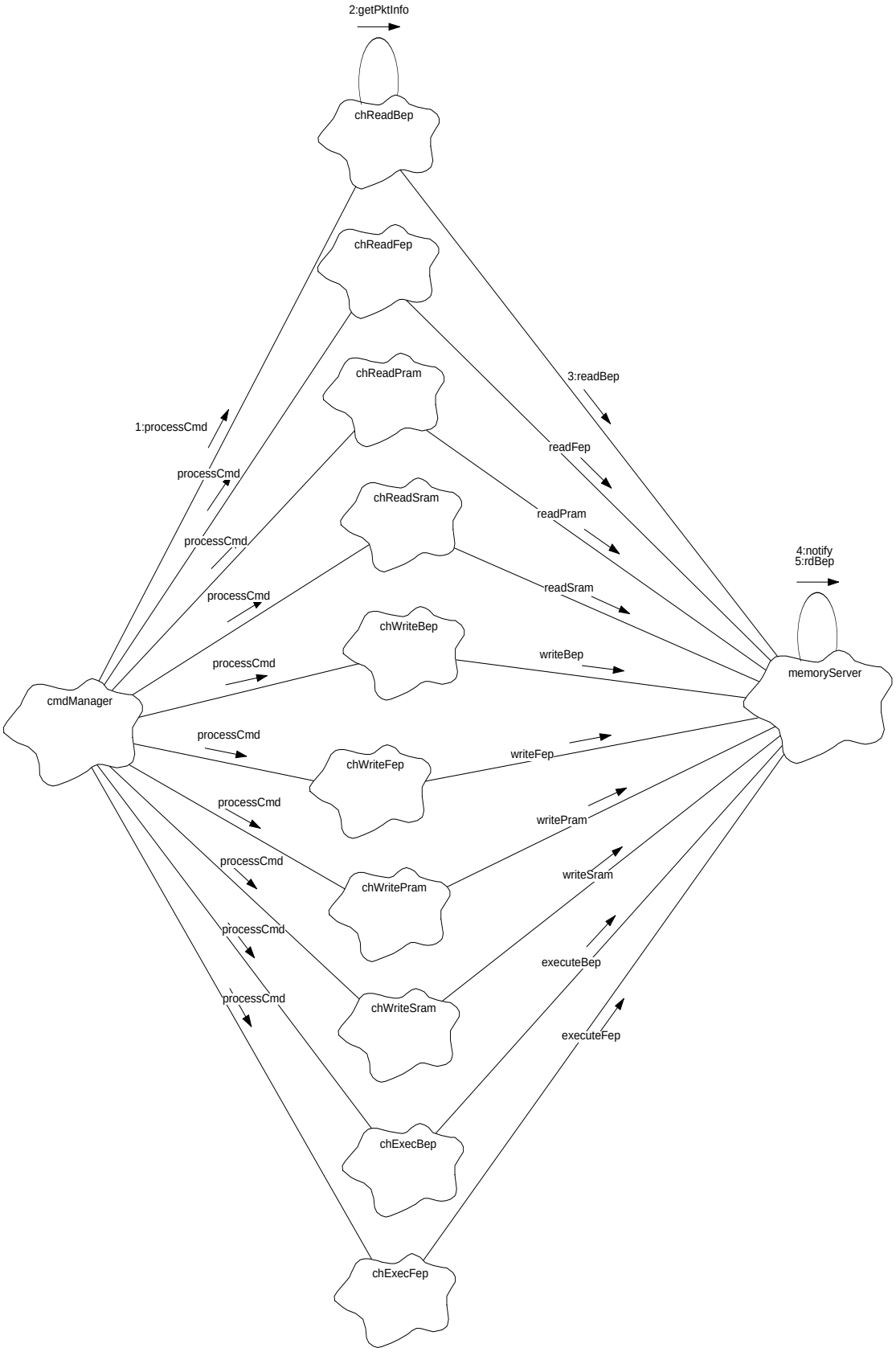


1. The *cmdManager* receives a command, and uses the packet’s opcode to select the appropriate handler (see Section 16.0 for more detail). The *cmdManager* then invokes the selected objects’s (in this case, *chLoadTimedExp*) processCmd() member function, passing the received command packet instance, *cmdPkt*.
2. *chLoadTimedExp*.processCmd() then obtains the packet’s data buffer and length by calling getPktInfo(), which is inherited from **CmdHandler**.
3. **CmdHandler**::getPktInfo() uses the command packet functions, *cmdPkt*.getDataAddress() and *cmdPkt*.getDataLength() to acquire the packet’s data buffer address and length.
4. *chLoadTimedExp*.processCmd() then attempts to obtain exclusive access to the Timed Exposure Parameter Block List by invoking *pblTimedExp*.waitForLock(). If the wait time-out expires, processCmd() returns a CMDRESULT_BUSY status to the Command Manager.
5. If processCmd() succeeds in locking the parameter block list, it instructs the list to copy the command packet parameters into the appropriate parameter block slot, using *pblTimedExp*.replaceBlock(). Once replaceBlock() returns, processCmd() relinquishes access to the parameter block list, using *pblTimedExp*.releaseLock(). If the earlier call to replaceBlock() succeeded, processCmd() returns CMDRESULT_OK. If it failed, it returns CMDRESULT_BAD_ARGUMENT.

17.5.2 Use 2:: Initiate Memory Dumps, Loads and Execution

Figure 70 illustrates the behaviors of the three commands directed toward the **MemoryServer**. Steps 1 - 5 illustrate the overall scenario for a “Read BEP Memory” command, from the point at which the *cmdManager* dispatches control of the *chReadBep* command handler instance, to the point at which the **MemoryServer** dispatches control to its private function, rdBep() to actually perform the memory read (NOTE: In order to clarify the overall behavior of the command handlers in association with the **MemoryServer**, this example violates strict encapsulation rules, and peeks a little bit into the internal implementation of the **MemoryServer**). The remaining objects in the diagram are not described step-by-step, but the overall behavior of each of the command handlers is the same at that for the “Read BEP Memory” example. They are shown in the diagram to illustrate the mapping each command handler instance to the corresponding binding function provided by the **MemoryServer**.

FIGURE 70. Memory Server Command Handling



1. The *cmdManager* receives a “Read BEP Memory” command, indexes *chReadBep* and invokes its processCmd() function.
2. **ChReadBep**::processCmd() then reads the command packet’s buffer address and length, using the inherited function, **CmdHander**::getPktInfo(). It then extracts source address and read length from the packet data buffer.
3. processCmd() invokes the **MemoryServer**’s *memoryServer*.readBep() function, passing the command identifier, the read source address and the number of 32-bit words to read.
4. The **MemoryServer** then attempts to gain exclusive access of its request buffer (not shown). If it succeeds, it stores the passed information and uses notify() to tell its task to execute the command, and immediately returns with CMDRESULT_OK. If *memoryServer* is busy, the request is dropped, and readBep() returns CMDRESULT_BUSY, which is then returned by *chReadBep.*processCmd(). If the arguments passed to the **MemoryServer** are invalid, it will return CMDRESULT_BAD_ARGUMENT.
5. At some later time, the waiting task portion of the **MemoryServer** will be scheduled to run and will eventually invoke its private rdBep() function to read the memory, format and send the read data to telemetry.

17.5.3 Use 3:: Start and Stop Science and DEA Housekeeping Runs

17.5.3.1 Handler Object Construction

Different types of “Start Science Run” and “Stop Science Run” command handlers are discriminated only by their command operation code and parameter block list reference. Figure 71 illustrates the overall initialization procedure used to construct command handlers which start and stop science runs within ACIS. The scenario description details the actions taken when constructing the “Start Timed Exposure Run” command handler. The scenario applies all of the remaining constructors, by varying only the command operation codes and science mode references for each distinct command handler object. Since “Start DEA Housekeeping Run” and “Stop DEA Housekeeping Run” command handlers each have only one instance, their construction is shown in the diagram, but is not directly explained in the accompanying scenario description.

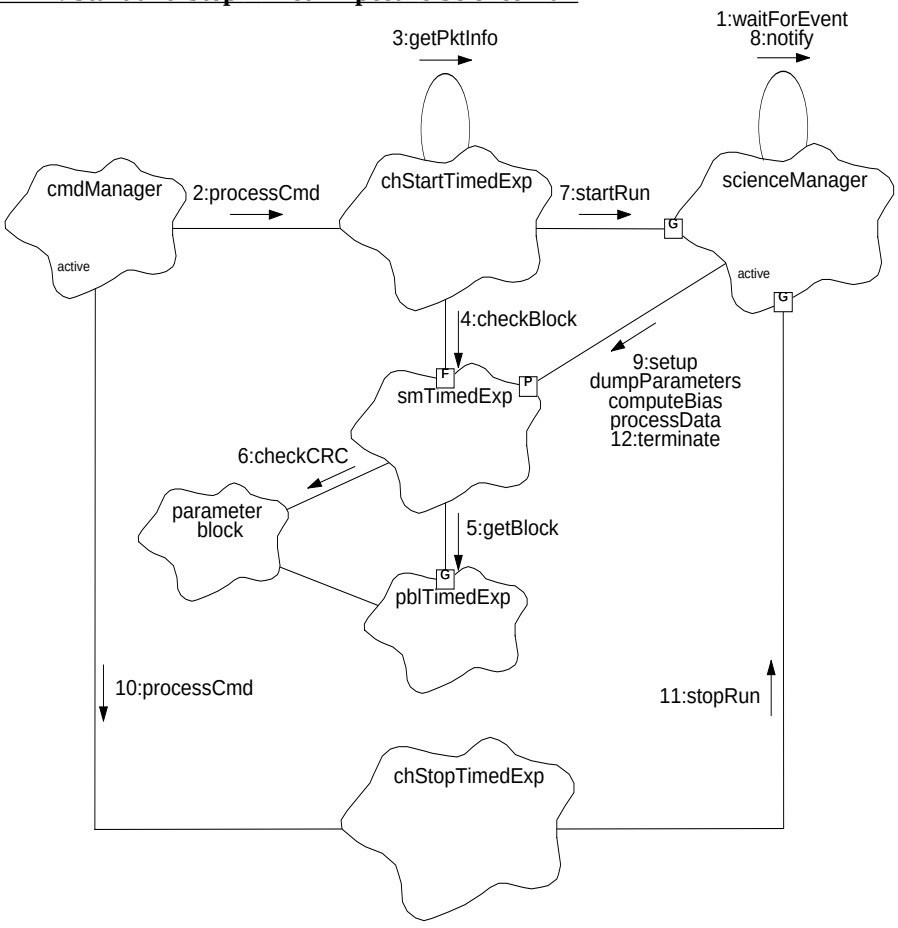
FIGURE 71. Start and Stop Run Command Handler Initialization

1. The global constructor list invokes the constructor for *chStartTimedExp*, passing CMDOP_START_TE as the command opcode, and a reference to *smTimedExp* as the science mode associated with *chStartTimedExp*.
2. *chStartTimedExp*'s constructor then invokes its parent constructor, **CmdHandler**::CmdHandler, passing the supplied command operation code. Once its parent is initialized, *chStartTimedExp* initializes its parameter block list pointer with the passed reference to *smTimedExp*.
3. The **CmdHandler** constructor invokes the **CmdManager** function, setHandler(), to install itself as the handler for the supplied command operation code.

17.5.3.2 Execution of Start and Stop Science Commands

Figure 72 illustrates the overall operation of a "Start Timed Exposure" and "Stop Timed Exposure" command sequence. The scenario for starting and stopping a Continuous Clocking Run is the same except the *chStartTimedExp*, *smTimedExp*, and *pblTimedExp* objects are replaced with the *chStartCc*, *smContClocking*, and *pblContClock* objects, respectively.

FIGURE 72. Start and Stop Timed Exposure Science Run

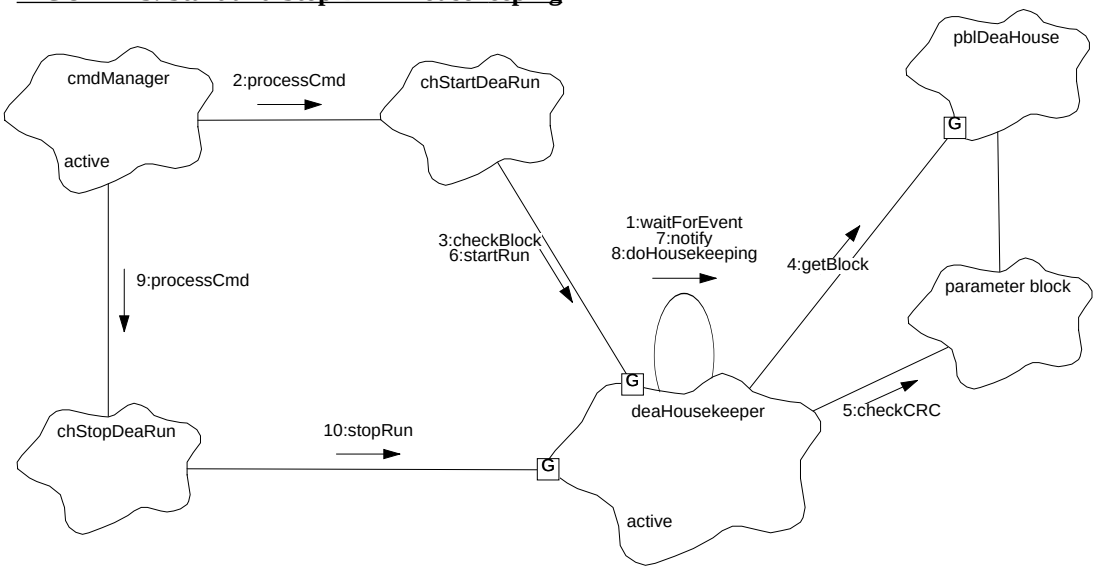


1. The *scienceManager* is initially idle, waiting for a command notification via **Task**::waitForEvent().
2. The *cmdManager* receives a "Start Timed Exposure Science Run" command packet, indexes and invokes *chStartTimedExp*.processCmd(), passing the command packet as an argument (not shown).
3. *chStartTimedExp*.processCmd() uses the inherited function, **CmdHandler**::getPktInfo() to get the address and length of the data portion of the packet. It then reads the parameter block id from the packets data area.
4. processCmd() then invokes *mode*->checkBlock() (effectively calling *smTimedExp*.checkBlock()) to verify the validity of the parameter block id and the integrity of the selected parameter block.
5. *smTimeExp*.checkBlock() calls *pblTimedExp*.getBlock() to retrieve a reference to the indexed parameter block.
6. If the block id is valid, *smTimedExp*.checkBlock() invokes the parameter block's checkCrc() function to ensure that it has not been corrupted. See TBD for a detailed description of the error behavior if either the parameter block id is invalid, or if the parameter block is corrupt.
7. If the return value from *smTimedExp*.checkBlock() indicates that the block exists and is not corrupted, processCmd() invokes the **ScienceManager**'s startRun() function to initiate the science run. If there is a problem with the parameter block, the handler logs the occurrence in software housekeeping (not shown), and uses the alternate supplied by *smTimedExp*.checkBlock(). If no alternate is specified, processCmd() does not call the **ScienceManager**. In either case, processCmd() returns the appropriate **CmdResult** (TBD), indicating the chosen action.
8. *scienceManager*.startRun() logs the passed parameter block identifier and science mode, and uses notify() to inform its task that a command is being requested.
9. Later, the task portion of the ScienceManager wakes up from waitForEvent() (due to the earlier notify()), and proceeds to start the science run, using the stored mode's setup(), dumpParameters(), computeBias(), processData() member functions.
10. The *cmdManager* receives a "Stop Timed Exposure Run" command, and invokes the indexed *chStopTimedExp*.processCmd() function.
11. *chStopTimedExp*.processCmd() then invokes *scienceManager*.stopRun(), which logs the stop request and notifies its task (not shown).
12. At some later time, the *scienceManager* task notices the state change request, and invokes the active mode's terminate() function. The *scienceManager* task then returns to its top loop, which calls waitForEvent() (see Step 1).

17.5.3.3 Execution of Start and Stop DEA Housekeeping Commands

Figure 73 illustrates the behavior of a Start and Stop DEA Run command sequence. The scenario is similar to that for science runs, except that the scenario invokes the **`DeaHousekeeper`** and that the housekeeper is responsible for interacting with the DEA Housekeeping Parameter Block List.

FIGURE 73. Start and Stop DEA Housekeeping



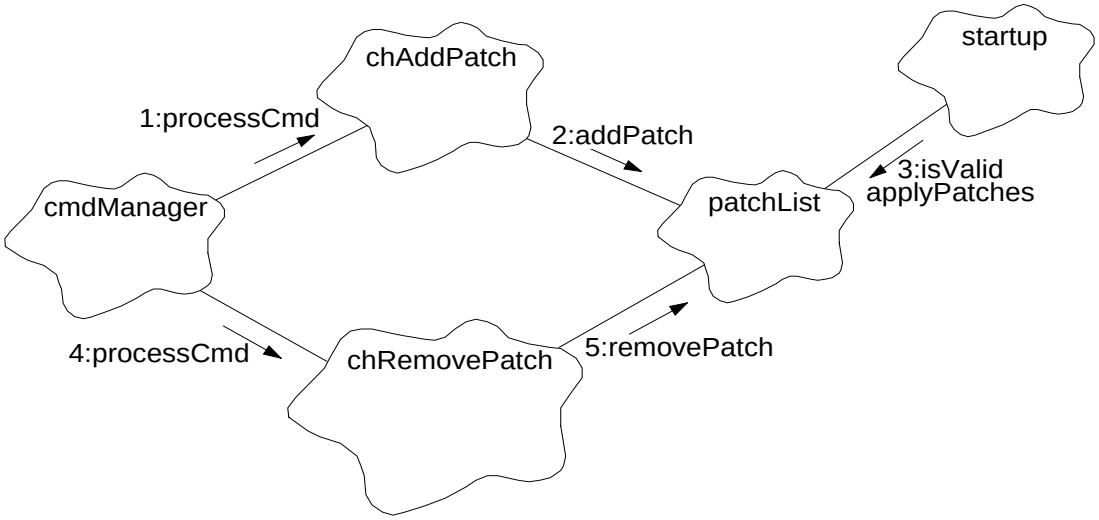
1. The *deaHousekeeper* is initially idle, waiting for notification of a command request, using `waitForEvent`().
2. The *cmdManager* receives a “Start DEA Housekeeping” command, and indexes and dispatches *chStartDeaHouse*.`processCmd`(), passing the received command packet (not shown).
3. *chStartDeaHouse*.`processCmd`() invokes the inherited function **`CmdHandler`**::`getPktInfo`() to obtain the packet’s data buffer and length (not shown). `processCmd`() then reads the parameter block id from the data area of the packet and invokes DEA housekeeper function, *deaHousekeeper*.`checkBlock`().
4. `checkBlock`() then invokes *pblDeaHouse*.`getBlock`() to obtain a reference to the identified parameter block. If the parameter block id is invalid, `checkBlock`() returns an error to `processCmd`() which then returns the appropriate error result to the *cmdManager*.
5. If the parameter block id is valid, `checkBlock`() invokes `checkCrc`() on the supplied parameter block, and returns whether or not the block is corrupted to `processCmd`().
6. If the parameter block is intact, `processCmd`() calls *deaHousekeeper*.`startRun`(). If the parameter block id is invalid, or of the block is corrupted, `processCmd`() returns the appropriate **`CmdResult`** (TBD), indicating the chosen action to its caller, but does not call *deaHousekeeper*.

7. *deaHousekeeper*.startRun() then logs the request information, and calls the inherited function, **Task**::notify() to wake up its task portion.
8. Eventually, the task portion of *deaHousekeeper* wakes up from its waitForEvent() call, and invokes its private doHousekeeping() function to start acquiring and sending DEA Housekeeping information.
9. Sometime later, the *cmdManager* receives a "Stop DEA Housekeeping" command, and indexes and invokes *chStopDeaRun*.processCmd().
10. *chStopDeaRun*.processCmd() then invokes *deaHousekeeper*.stopRun(). stopRun() then logs the request and notifies the housekeeper's task (not shown). Later on, the task detects the stop request and returns to its top task loop, which calls waitForEvent() (see Step 1).

17.5.4 Use 4:: Load and Remove Patches

Figure 74 illustrates the handling of "Add Patch" and "Remove Patch" commands.

FIGURE 74. Add and Remove Patch Command Handling



1. The *cmdManager* receives an "Add Patch" command. It indexes and dispatches *chAddPatch*.processCmd(), passing the command packet.
2. *chAddPatch*.processCmd() uses the inherited **CmdHandler**::getPktInfo() (not shown) to get the packet's data address and length. It passes the obtained patch information to *patchList*.addPatch(), which then installs the patch in the system's patch list.
3. The instrument receives a hardware reset command, reloads its code and data from ROM into RAM and invokes the loaded start-up procedure. The start-up procedure then invokes *patchList*.isValid() to verify the integrity of the patch list. If the list is intact, it invokes *patchList*.applyPatches(), which uses information in the list to

modify the code and data in RAM. Once this is complete, start-up proceeds to continue its initialization and eventually brings up the now temporarily modified instrument software.

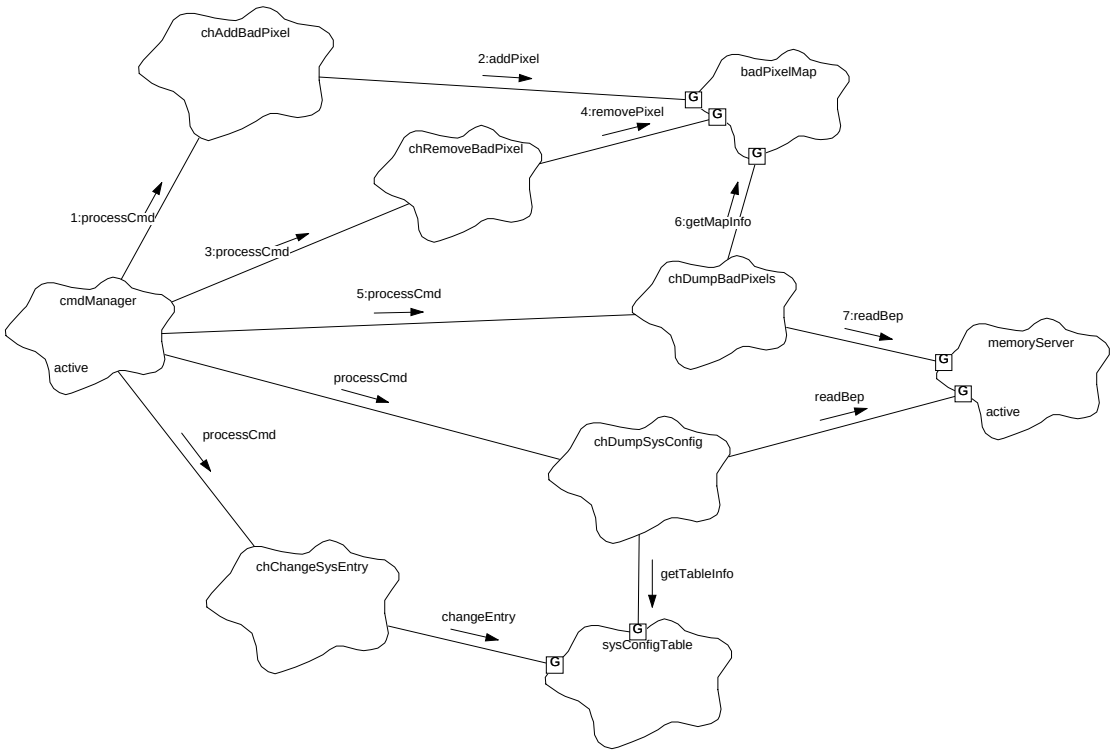
4. The *cmdManager* receives a command to remove a patch. It indexes and invokes *chRemovePatch*.processCmd().
5. *chRemovePatch*.processCmd() uses **CmdHandler**::getPktInfo() to obtain the patch identification information, and calls *patchList*.removePatch() to eliminate the identified patch from the list. When a subsequent hardware reset is sent, the original code and data are reloaded from ROM into RAM, and the start-up code installs its patches, minus the one(s) just removed by the previous command.

17.5.5 Use 5:: Modify and Dump System Configuration Settings

Figure 75 illustrates the operations used to add, remove and dump bad pixels within ACIS. The illustration also shows how the system configuration settings are modified and dumped, but this is not described in the detailed scenario description. Not shown are the operations to add, remove and dump bad columns from either the Timed Exposure Bad Column Map, nor the Continuous Clocking Bad Column Map. The overall sequences for these two maps are the same as the sequence from Bad Pixels, just with different objects participating. Not shown is that the Timed Exposure and Continuous Clocking Bad Column map commands use different instances of the same class, which are discriminated during construction by a passed command opcode and a reference to their respective Bad Column Maps.

NOTE: Since some structures, including the bad pixel map, may be larger than will fit into a single telemetry packet, a design decision has been made to use the **MemoryServer** to dump the system configuration data structures. The effect is that the information will reach the ground without specific format information contained with the telemetry packet. It is assumed that the ground can use the dump address and memory map, or can use the command packet identifier to determine what was dumped and why.

FIGURE 75. System Configuration Command Handling



1. The *cmdManager* receives a “Add Bad Pixel” command, and calls *chAddBadPixel*.processCmd(), passing the received command packet.
2. *chAddBadPixel*.processCmd() uses **CmdHandler**::getPktInfo() to read the data portion of the packet (not shown). It then passes the bad pixel information onto *badPixelMap*.addPixel(), which stores the information.
3. The *cmdManager* later receives a “Remove Bad Pixel” command, and invokes *chRemoveBadPixel*.processCmd().
4. *chRemoveBadPixel*.processCmd() extracts the pixel identification information, and passes it onto *badPixelMap*.removePixel(), to eliminate the identified pixel from the system’s list.
5. The *cmdManager* receives a “Dump Bad Pixel” command, and invokes *chDumpBadPixel*.processCmd().
6. *chDumpBadPixel*.processCmd() calls *badPixelMap*.getMapInfo() to obtain the base address and length of the current bad pixel map.
7. *chDumpBadPixel*.processCmd() then calls *memoryServer*.readBep(), passing the address and length of the map. If the *memoryServer* is not busy, it will register a request, and proceed to dump the indicated region into telemetry once its task portion is scheduled to run. If the *memoryServer* is busy (or if the arguments were invalid), it will return an error, which processCmd() passes back to the *cmdManager*.

17.6 Class ChAddBadCol

Documentation:

This class handles commands which add columns to one of the system's bad column maps, either the Timed Exposure Bad Column Map, or the Continuous Clocking Bad Column Map. Each instance of this class is associated to a single bad column map and command operation code when it is constructed. There is one instance of this class for use with the Timed Exposure Bad Column Map, and another instance which modifies the Continuous Clocking Bad Column Map.

Export Control: Public

Cardinality: 2

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`BadColumnMap`

Public Interface:

Operations: `ChAddBadCol()`
`processCmd()`

Private Interface:

Has-A Relationships:

`BadColumnMap&` *`map`*:: This is a reference to the bad column map maintained by this instance. It is initialized upon construction by the passed *`badColMap`* constructor argument.

Concurrency: Synchronous

Persistence: Persistent

17.6.1 ChAddBadCol()

Public member of: **ChAddBadCol**

Arguments:

CmdOpcode *opcode*
BadColumnMap& *badColMap*

Documentation:

This constructor initializes an instance of this class, passing opcode to its parent CmdHandler() constructor, which associates the instance with the particular *opcode*, and initializing its bad column map to the passed *badColMap*.

Concurrency: Sequential

17.6.2 processCmd()

Public member of: **ChAddBadCol**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function adds bad columns to the instance's bad column map, map. *pkt* points to the command packet containing the columns to add to the list. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_TABLE_FULL

Semantics:

Use **CmdHandler**::getPktInfo() to get access to *pkt*'s data buffer and length, cast the retrieved pointer to the bad pixel info structure. For each pixel contained within the buffer, call *map*.addColumn().

Concurrency: Synchronous

17.7 Class ChAddBadPixel

Documentation:

This class is responsible for handling commands which add pixels to the instrument's bad pixel map.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`BadPixelMap`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.7.1 processCmd()

Public member of: **ChAddBadPixel**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function handles requests to add pixels to the bad pixel map. *pkt* points to the command packet containing the pixels to add to the list. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_TABLE_FULL

Semantics:

Use **CmdHandler**::getPktInfo() to get access to *pkt*'s data buffer and length, cast the retrieved pointer to the bad pixel info structure. For each pixel contained within the buffer, call *badPixelMap*.addPixel().

Concurrency: Synchronous

17.8 Class ChAddPatch

Documentation:

This command handles commands which add patches to the system's patch list.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`PatchList`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.8.1 processCmd()

Public member of: **ChAddPatch**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function handles commands to add patches to the system patch list. *pkt* is a pointer to the command packet containing the patch to add. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_TABLE_FULL

Semantics:

Use **CmdHandler**::getPktInfo() to get *pkt*'s data buffer and length, and cast to a patch info structure. For each patch in the packet, call *patchList*.addPatch() to install the patch into the list.

Postconditions:

The patch will be applied to code/data loaded from ROM to RAM during start-up after the next commanded reset (watchdog resets do not apply patches).

Concurrency: Synchronous

17.9 Class ChChangeSysEntry

Documentation:

This class is responsible for handling commands which modify one or more system configuration entries.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`SysConfigTable`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.9.1 processCmd()

Public member of: **ChChangeSysEntry**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function handles commands which modify system configuration entry values. *pkt* points to the requesting command packet, and contains the entry modifications. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT

Semantics:

Use **CmdHandler**::getPktInfo() to get the *pkt*'s data buffer and length, cast the result to entry info. For each entry in the packet, call *sysConfigTable*.changeEntry() to modify the value associated with the specified entry.

Concurrency: Synchronous

17.10 Class ChDumpBadCol

Documentation:

This class handles commands to dump the contents of one of the system's bad column maps to telemetry. A given instance of this class is associated to a command operation code and a given bad column map by the class's constructor.

Export Control: Public

Cardinality: 2

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`MemoryServer`
`BadColumnMap`

Public Interface:

Operations: `ChDumpBadCol()`
`processCmd()`

Private Interface:

Has-A Relationships:

`BadColumnMap&` *`map`*:: This is a reference to the bad column map maintained by this handler instance. It is initialized by the class constructor using the passed *`badColMap`* reference.

Concurrency: Synchronous

Persistence: Persistent

17.10.1 ChDumpBadCol()

Public member of: **ChDumpBadCol**

Arguments:

CmdOpcode *opcode*
BadColumnMap& *badColMap*

Documentation:

This constructor initializes an instance of this class, passing *opcode* to its parent CmdHandler() constructor, which associates the instance with the particular opcode, and initializing its bad column map to the passed *badColMap*.

Concurrency: Sequential

17.10.2 processCmd()

Public member of: **ChDumpBadCol**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function initiates a dump of the contents of the instance's bad column map. *pkt* is a pointer to the command packet requesting the dump, and is not used by this function. This function relies on the **MemoryServer** to perform the actual dump. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_TABLE_EMPTY
CMDRESULT_BUSY

Semantics:

Call *map*.getTableInfo() to get the bad column map's address and length. If the length is not zero, call *memoryServer*.readBep() to initiate a dump of the map.

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the region of memory containing the bad column map. Once the *memoryServer*'s task is scheduled to run, the dump will start. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.11 Class ChDumpBadPixels

Documentation:

This class handles commands which request a dump of the bad pixel map to telemetry.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`MemoryServer`
`BadPixelMap`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.11.1 processCmd()

Public member of: **ChDumpBadPixels**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function initiates a dump of the bad pixel map to telemetry. *pkt* is a pointer to the command packet requesting the dump, and is not used by this function. This function relies on the **MemoryServer** to perform the actual dump. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_TABLE_EMPTY
CMDRESULT_BUSY

Semantics:

Call *badPixelMap*.getTableInfo() to get the bad pixel map's address and length. If the length is not zero, call *memoryServer*.readBep() to initiate a dump of the map.

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the region of memory containing the bad pixel map. Once the *memoryServer*'s task is scheduled to run, the dump will start. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.12 Class ChDumpSysConfig

Documentation:

This class is responsible for handling commands which request a dump of the entire system configuration table.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`SysConfigTable`
`MemoryServer`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.12.1 processCmd()

Public member of: **ChDumpSysConfig**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function handles commands which request a dump of the system configuration entries to telemetry. *pkt* points to the command packet making the request, but is unused by this function. This function relies on the **MemoryServer** to perform the actual dump. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_BUSY

Semantics:

Call *sysConfigTable*.getTableInfo() to get the entry table's address and length. Call *memoryServer*.readBep() to initiate a dump of the map.

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the region of memory containing the system configuration. Once the *memoryServer*'s task is scheduled to run, the dump will start. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.13 Class ChExecBep

Documentation:

This class is responsible for handling commands which request the execution of a subroutine located within the Back End Processor's memory.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.13.1 processCmd()

Public member of: **ChExecBep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s executeBep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to execute BEP information structure, and call *memoryServer*.executeBep().

Postconditions:

If successful, the *memoryServer* will have logged a request to execute the function and telemeter the result. Once the *memoryServer*'s task is scheduled to run, the function will be called and the *memoryServer* will remain busy until the call completes. Once the call returns, the *memoryServer* will telemeter the returned value.

Concurrency: Synchronous

17.14 Class ChExecFep

Documentation:

This class handles commands to execute a function located in one of the Front End Processor's memory, and telemeter the return value.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.14.1 processCmd()

Public member of: **ChExecFep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s executeFep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to execute FEP information structure, and call *memoryServer*.executeFep().

Postconditions:

If successful, the *memoryServer* will have logged a request to execute the function on the specified Front End Processor, and telemeter the result. Once the *memoryServer*'s task is scheduled to run, it will dispatch the request to the Front End and the *memoryServer* will remain busy until the request completes. Once the request completes, the *memoryServer* will telemeter the returned value.

Concurrency: Synchronous

17.15 Class ChLoadBlk

Documentation:

This class provides the command handler for all "load parameter block" commands. The association between a given type of block and the command opcode is determined by the constructor for this class.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`CmdHandler`**

Public Interface:

Operations:
```
ChLoadBlk()
processCmd()
```

Private Interface:

Has-A Relationships:

`PblockList&` *`pblockList`*: This is a reference to the parameter block list managed by the instance.

`const unsigned` *`loadTimeout`*: This constant determines the timeout period, in clock ticks (1/10 second), that the command handler uses to wait for access to the parameter block list. This value must be less than 200ms to ensure that the handler returns in time to process the next command.

Concurrency: Synchronous

Persistence: Persistent

17.15.1 ChLoadBlk()

Public member of: **ChLoadBlk**

Arguments:

CmdOpcode *opcode*
PblockList& *blocklist*

Documentation:

This is the constructor for all Load Parameter Block command handlers. *opcode* specifies the command packet opcode used for the command, and *blocklist* is a reference to the parameter block list modified by the handler.

Concurrency: Sequential

17.15.2 processCmd()

Public member of: **ChLoadBlk**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function processes all "load parameter block" commands, installing the parameter block contained in *cmdpkt*. If the load is successful, this function returns CMDRESULT_OK. If the command cannot obtain access to the parameter block, it returns CMDRESULT_BUSY. If the parameter block contained within the command has been corrupted, it returns CMDRESULT_CORRUPT.

Concurrency: Guarded

17.16 Class ChReadBep

Documentation:

This class handles commands to read and telemeter the contents of the Back End Processor's memory.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`MemoryServer`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.16.1 processCmd()

Public member of: **ChReadBep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s readBep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to read BEP memory information structure, and call *memoryServer*.readBep().

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the specified region of memory. Once the *memoryServer*'s task is scheduled to run, the dump will start. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.17 Class ChReadFep

Documentation:

This class handles commands to read and telemeter the contents of one the Front End Processor's memory.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.17.1 processCmd()

Public member of: **ChReadFep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s readFep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to read FEP memory information structure, and call *memoryServer*.readFep().

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the specified region of memory from the specified Front End Processor. Once the *memoryServer*'s task is scheduled to run, it will start performing the dump. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.18 Class ChReadPram

Documentation:

This class handles commands to read and telemeter the contents of the one of the DEA CCD Controller's Program RAM (PRAM).

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.18.1 processCmd()

Public member of: **ChReadPram**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s readPram() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to read PRAM memory information structure, and call *memoryServer*.readPram().

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the specified region of memory from the specified DEA CCD controller board's Program RAM. Once the *memoryServer*'s task is scheduled to run, it will start performing the dump. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.19 Class ChReadSram

Documentation:

This class handles commands to read and telemeter the contents of the one of the DEA CCD Controller's Sequencer RAM (SRAM).

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.19.1 processCmd()

Public member of: **ChReadSram**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s readSram() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to read SRAM memory information structure, and call *memoryServer*.readSram().

Postconditions:

If successful, the *memoryServer* will have logged a request to read and telemeter the specified region of memory from the specified DEA CCD controller board's Sequencer RAM. Once the *memoryServer*'s task is scheduled to run, it will start performing the dump. The *memoryServer* will remain busy until the dump completes.

Concurrency: Synchronous

17.20 Class ChRemoveBadCol

Documentation:

This class handles commands to remove pixels from one of the system's bad column maps. A given instance of this class is associated to a command operation code and a given bad column map by the class's constructor.

Export Control: Public

Cardinality: 2

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`BadColumnMap`

Public Interface:

Operations: `ChRemoveBadCol()`
`processCmd()`

Private Interface:

Has-A Relationships:

`BadColumnMap&` *`map`*:: This is a reference to the bad column map associated with this handler instance. It is initialized during construction using the *`badColMap`* argument.

Concurrency: Synchronous

Persistence: Persistent

17.20.1 ChRemoveBadCol()

Public member of: **ChRemoveBadCol**

Arguments:

CmdOpcode *opcode*
BadColumnMap& *badColMap*

Documentation:

This constructor initializes the handler instance, passing *opcode* to the parent constructor CmdHandler() which associates the instance with the command, and initializing *map* with the passed *badColMap* argument.

Concurrency: Sequential

17.20.2 processCmd()

Public member of: **ChRemoveBadCol**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function removes pixels from the bad column map managed by this instance, *map*. *pkt* is the command packet containing a list of columns to remove. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT

Semantics:

Use **CmdHandler**::getPktInfo() to get *pkt*'s data buffer and length, cast to the bad column list info type. For each column listed, call *map*.removeColumn() to eliminate the column from the list.

Concurrency: Synchronous

17.21 Class ChRemoveBadPixel

Documentation:

This class handles commands to remove pixels from the bad pixel map.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`BadPixelMap`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.21.1 processCmd()

Public member of: **ChRemoveBadPixel**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function removes pixels from the system's bad pixel map. *pkt* is the command packet containing a list of pixels to remove. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT

Semantics:

Use **CmdHandler**::getPktInfo() to get *pkt*'s data buffer and length, cast to the bad pixel list info type. For each pixel listed, call *badPixelMap*.removePixel() to eliminate the pixel from the list.

Concurrency: Synchronous

17.22 Class ChRemovePatch

Documentation:

This class handles commands which remove patches from the system patch list.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`PatchList`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.22.1 processCmd()

Public member of: **ChRemovePatch**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function removes patches from the system patch list. *pkt* points to the command packet containing the patch identifiers to be removed. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATAPTR
CMDRESULT_INVALID_DATACNT
CMDRESULT_BAD_ARGUMENT

Semantics:

Call **CmdHandler**::getPktInfo() to get *pkt*'s data pointer and length, cast to remove patch info structure. For each patch identifier in the list, call *patchList*.removePatch() to remove the patch from the list.

Postconditions:

Upon the next commanded reset, the code and data are reloaded from ROM to RAM, and the remaining patches in the patch list are applied (sans the removed patches).

Concurrency: Synchronous

17.23 Class ChStartDeaRun

Documentation:

This class is responsible for handling commands which start DEA Housekeeping runs.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`DeaHousekeeper`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.23.1 processCmd()

Public member of: **ChStartDeaRun**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function processes requests to start a DEA Housekeeping run. *cmdpkt* points to the requesting command packet, and contains the block id to use for the run. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_CORRUPT_DEFAULT
CMDRESULT_CORRUPT_IDLE

Semantics:

Use **CmdHandler**::getPktInfo() to get data buffer and length, cast to DEA Info structure and get block id. Pass block id to *deaHousekeeper*.checkBlock(). If it succeeds, call *deaHousekeeper*.startRun() and return CMDRESULT_OK. If it fails, but *alternate* is valid, still call *deaManager*.startRun() but return CMDRESULT_CORRUPT_DEFAULT. If the alternate is invalid, just return CMDRESULT_CORRUPT_IDLE without starting a run.

Postconditions:

If the parameters are ok, or a default was selected, the *deaHousekeeper* will have logged the request information. Once its task portion is scheduled to run (and after it cleans up any pending operations), the housekeeper will start acquiring and sending the housekeeping information to telemetry.

Concurrency: Synchronous

17.24 Class ChStartSciRun

Documentation:

This class handles all Start Science Run commands. The constructor for this class provides the association between the start command for a particular mode, and the science mode within the instrument.

Export Control: Public

Cardinality: 4

Hierarchy:

Superclasses: **`CmdHandler`**

Public Uses:

`ScienceMode`

Implementation Uses:

`ScienceManager`

Public Interface:

Operations: `ChStartSciRun()`
`processCmd()`

Private Interface:

Has-A Relationships:

`ScienceMode&` *`mode`*:: This is reference to the science mode started by this command handler instance. *`mode`* is assigned during construction of the handler. When starting a run, *`mode`* is passed to the **`ScienceManager`** to instruct it which science mode to run.

`const Boolean` *`biasonly`*:: This flag indicates whether or not this command handler starts complete science runs, or bias only runs. If this flag is *`BoolFalse`*, complete science runs are dispatched. If this flag is *`BoolTrue`*, bias-only runs are initiated. This flag is initialized by the constructor.

Concurrency: Synchronous

Persistence: Persistent

17.24.1 ChStartSciRun()

Public member of: **ChStartSciRun**

Arguments:

CmdOpcode *opcode*
ScienceMode& *mode*
Boolean *biasOnly*

Documentation:

This is the constructor for a science run command handler. *opcode* is the command operation handled by the instance being constructed, *scienceMode* is a reference to the science mode started by this handler, and *biasOnly* is a flag indicating if this command is handling compute bias requests (rather than full science runs). If *biasOnly* is *BoolFalse*, the handler will initiate complete science runs. If *biasOnly* is *BoolTrue*, then the handler will initiate bias computations only.

Semantics:

Initialize parent CmdManager, passing *opcode*. Set local variable, *mode*, to point to the passed *scienceMode*, and copy *biasOnly* to the private variable, *biasonly*.

Concurrency: Sequential

17.24.2 processCmd()

Public member of: **ChStartSciRun**

Return Class: **CmdResult**

Arguments:

CmdPkt* *cmdpkt*

Documentation:

This function handles commands which initiate science runs. *cmdpkt* is a pointer to the requesting command packet, and contains the parameter block id to use for the science run. This function returns one of the following:

CMDRESULT_OK
CMDRESULT_BAD_DATACNT
CMDRESULT_BAD_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_CORRUPT_DEFAULT
CMDRESULT_CORRUPT_IDLE

Semantics:

Use **CmdHandler**::getPktInfo() to get data buffer, cast to run info structure and extract block id. Pass block id to *mode*.checkBlock(). If checkBlock() returns *BoolTrue*, invoke *scienceManager*.startRun(), and return CMDRESULT_OK. If checkBlock() returns *BoolFalse*, but *alternate* is valid, pass *alternate* to *scienceManager* but return CMDRESULT_CORRUPT_DEFAULT. If the supplied *alternate* is invalid, do not invoke *scienceManager*, just return CMDRESULT_CORRUPT_IDLE.

Postconditions:

If parameters are ok, or if a default mode was selected, *scienceManager* will have recorded a request to perform a science run. Once its task is scheduled (and it cleans up from its previous operations), the requested run will be started. If the parameters are invalid, and no default is supplied, the *scienceManager* is left alone.

Concurrency: Synchronous

17.25 Class ChStopDeaRun

Documentation:

This class is responsible for handling command requesting that DEA Housekeeping operations stop.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`DeaHousekeeper`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.25.1 processCmd()

Public member of: **ChStopDeaRun**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function processes requests to stop a DEA housekeeping run. *pkt* points to the requesting command packet. This function checks the passed command, reports discrepencies to software housekeeping, and then stops the run. This function always returns CMDRESULT_OK.

Semantics:

Call *deaHousekeeping*.stopRun().

Concurrency: Synchronous

17.26 Class ChStopSciRun

Documentation:

This class handles all science stop run commands. Each instance is used to handle a particular “stop science run” request.

Export Control: Public

Cardinality: 2

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`ScienceManager`

Public Interface:

Operations:
`ChStopSciRun()`
`processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.26.1 ChStopSciRun()

Public member of: **ChStopSciRun**

Arguments:

CmdOpcode *opcode*

Documentation:

This constructor initializes the state of the handler instance, and using its parents constructor, `CmdHandler`(), associates the instance with the command `opcode` used to terminate a particular science mode.

Semantics:

Initializer calls `CmdHandler`() passing `opcode` to associate this instance with a particular “stop run” operation code.

Concurrency: Sequential

17.26.2 processCmd()

Public member of: **ChStopSciRun**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function handles requests to stop a running science mode. *`cmdpkt`* points to the requesting command packet. This function checks the contents of *`cmdpkt`* with the current science state, reports discrepencies to software housekeeping, and stops the run. This function always returns CMDRESULT_OK.

Semantics:

Call *`scienceManager`*.`stopRun`().

Concurrency: Synchronous

17.27 Class ChWriteBep

Documentation:

This class handles commands to write the contents of the Back End Processor's memory.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.27.1 processCmd()

Public member of: **ChWriteBep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s writeBep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to write BEP memory information structure, and call *memoryServer*.writeBep().

Postconditions:

If successful, the *memoryServer* will have logged a request to write the specified region of memory. Once the *memoryServer*'s task is scheduled to run, it will copy the registered data (or code) to the specified memory address. The *memoryServer* will remain busy until the write completes.

Concurrency: Synchronous

17.28 Class ChWriteFep

Documentation:

This class handles commands to write the contents of one of the Front End Processors' memory.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses: **`MemoryServer`**

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.28.1 processCmd()

Public member of: **ChWriteFep**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s writeFep() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to write FEP memory information structure, and call *memoryServer*.writeFep().

Postconditions:

If successful, the *memoryServer* will have logged a request to write the specified region of memory of the identified Front End Processor. Once the *memoryServer*'s task is scheduled to run, it will request that the registered data (or code) to be copied to the specified memory address on the FEP. The *memoryServer* will remain busy until the write completes.

Concurrency: Synchronous

17.29 Class ChWritePram

Documentation:

This class handles commands to write the contents of one of the DEA CCD Controllers' Program RAM (PRAM).

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`MemoryServer`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.29.1 processCmd()

Public member of: **ChWritePram**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the MemoryServer's writePram() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to write PRAM memory information structure, and call *memoryServer*.writePram().

Postconditions:

If successful, the *memoryServer* will have logged a request to write the specified region of memory of the identified DEA CCD Controller's Program RAM. Once the *memoryServer*'s task is scheduled to run, it will request that the registered data (or code) to be copied to the specified memory location within in the DEA controller's PRAM. The *memoryServer* will remain busy until the write completes.

Concurrency: Synchronous

17.30 Class ChWriteSram

Documentation:

This class handles commands to write the contents of one of the DEA CCD Controllers' Sequencer RAM (SRAM).

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`CmdHandler`**

Implementation Uses:

`MemoryServer`

Public Interface:

Operations: `processCmd()`

Concurrency: Synchronous

Persistence: Persistent

17.30.1 processCmd()

Public member of: **ChWriteSram**

Return Class: **CmdResult**

Arguments:

const CmdPkt* *pkt*

Documentation:

This function interprets the contents of the command packet, *pkt*, and passes the extracted parameters to the **MemoryServer**'s writeSram() binding function. This function returns the following:

CMDRESULT_OK
CMDRESULT_INVALID_DATACNT
CMDRESULT_INVALID_DATAPTR
CMDRESULT_BAD_ARGUMENT
CMDRESULT_BUSY

Semantics:

Call **CmdHandler**::*getPktInfo()* to get *pkt*'s data buffer and length. Cast to write SRAM memory information structure, and call *memoryServer*.writeSram().

Postconditions:

If successful, the *memoryServer* will have logged a request to write the specified region of memory of the identified DEA CCD Controller's Sequencer RAM. Once the *memoryServer*'s task is scheduled to run, it will request that the registered data (or code) to be copied to the specified memory location within in the DEA controller's SRAM. The *memoryServer* will remain busy until the write completes.

Concurrency: Synchronous

18.0 Telemetry Management Classes (36-53215 A+)

18.1 Purpose

The purpose of the Telemetry Management system is to build and send a time-ordered list of telemetry packets to the instrument's telemetry hardware.

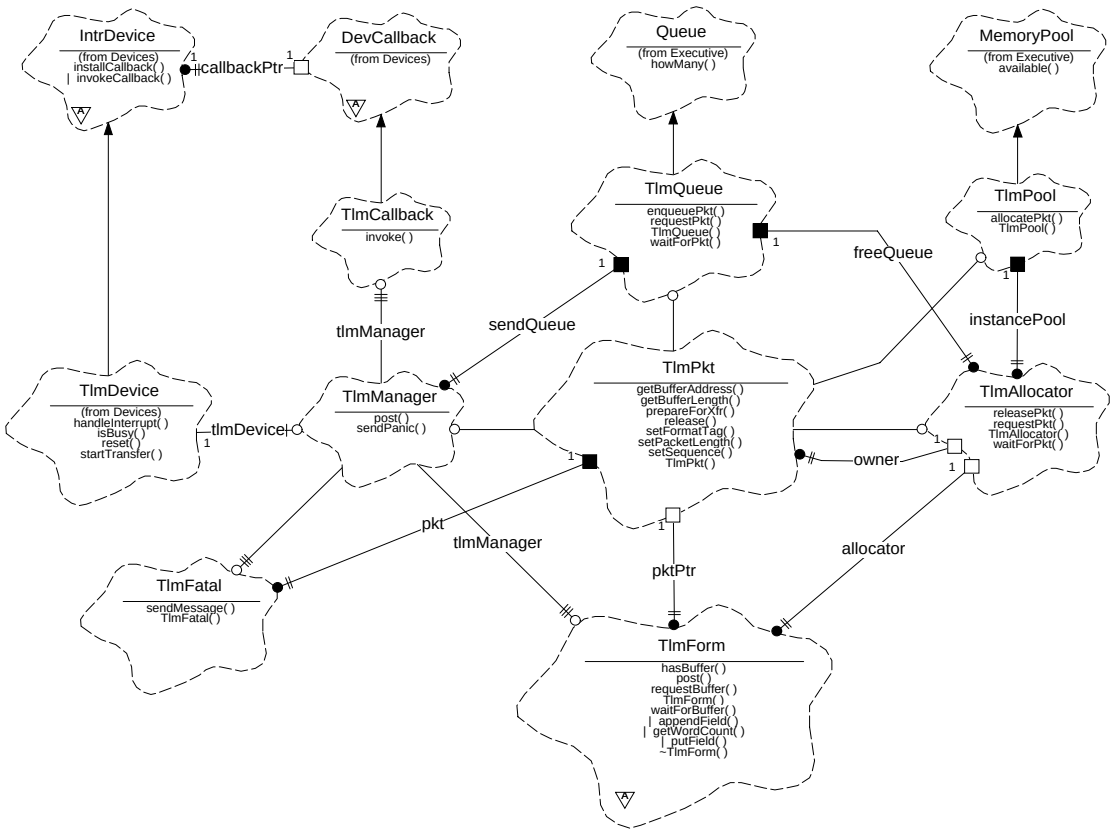
18.2 Uses

Use 1:: Allocate telemetry buffers and control structures
Use 2:: Manage the ordered transfer of telemetry information to the instrument hardware
Use 3:: Provide support for a final fatal error telemetry message prior to controlled crashes of the instrument software.

18.3 Organization

Figure 76 illustrates the top-level classes and their relationships involved in processing telemetry.

FIGURE 76. Telemetry Management Class Relationships



TlmManager - The **TlmManager** class is responsible for maintaining a queue of telemetry packets (see **TlmQueue** and **TlmPkt**) to feed to the telemetry device (***Devices***::**TlmDevice**).

TlmPkt - The **TlmPkt** class is responsible for managing the raw telemetry buffer located in Single Event Upset (SEU) vulnerable RAM, and for maintaining a small amount of critical packet header information within SEU immune RAM. This class serves as the container for telemetry information within the ACIS software.

TlmAllocator - The **TlmAllocator** class acts as a memory manager for a group of telemetry packet buffer types. It maintains a collection of equally sized **TlmPkt** instances. The allocator class uses the **TlmPool** class to obtain a set of **TlmPkt** instances during start-up, and then uses a **TlmQueue** to manage these instances at run-time.

TlmQueue - The **TlmQueue** class is a subclass of ***Executive***::**Queue**. It serves as a type-safe interface to a **Queue** instance which maintains an first-in/first-out list of pointers to **TlmPkt** instances. This class is used by the **TlmAllocator** to maintain its list of free telemetry packets. It is also used by the **TlmManager** to maintain a list of active telemetry packets awaiting transfer out of the instrument.

TlmPool - The **TlmPool** class is a subclass of ***Executive***::**MemoryPool**. It serves as a type-safe interface to a **MemoryPool** instance which contains the memory buffers used for **TlmPkt** instances. This class is used by the **TlmAllocator** during start-up to reserve memory for a set of **TlmPkt** buffers. The intent of this class is to provide the ability to patch the number or size of **TlmPkts** within a given pool instance, if the unlikely need arises after launch.

TlmForm -The **TlmForm** class is used as a base-class for the different telemetry packet formats. It and its subclasses are responsible for providing the interface functions needed to allocate telemetry packets, format the contents of the packets, and post the packets to the **TlmManager** for transfer out of the instrument. By convention, the name of all subclasses of **TlmForm** shall be prefixed with the abbreviation, "**Tf**" (i.e. a command echo telemetry format class would be called **Tf_Command_Echo**).

TlmDevice - The ***Devices***::**TlmDevice** is responsible for commanding the hardware to transfer a telemetry packet. It is a subclass of ***Devices***::**IntrDevice**, and is provided by the ***Devices*** class category. It is described in Section 9.0 .

TlmCallback - The **TlmCallback** class is a subclass of ***Devices***::**IntrCallback**. It is used by the **TlmManager** to obtain control during telemetry interrupt processing.

TlmFatal - The **TlmFatal** class is responsible for formatting fatal error messages. Due to the special nature of fatal error messages, this class is implemented independently of the telemetry format class, **TlmForm**.

18.4 Telemetry Processing Assumptions and Restrictions

18.4.1 Transfer buffers and packet formats

The hardware requires that all telemetry transfers must be from uncached RAM, must be less than or equal to 8192 bytes in size, must be aligned to a 4-byte boundary, and must not cross an 8192 byte boundary. In ACIS, this memory is vulnerable to Single-Event Upsets (SEU). In order to minimize the possibility of an upset corrupting crucial buffer management information, or important packet header information, the ACIS software maintains this information in its data cache space, which uses SEU immune RAM. This information is contained within a **`TlmPkt`** class instance. Each **`TlmPkt`** instance has a pointer to a packet transfer buffer within uncached RAM. In general, memory for **`TlmPkt`** instances are reserved using **`TlmPool`** instances (which use ***`Nucleus RTX`*** Memory Partitions), and lists of **`TlmPkt`**'s are maintained using **`TlmQueue`** instances (which use ***`Nucleus RTX`*** Queues).

During operation, all telemetry information except for the packet headers is written directly into a packet's transfer buffer by a **`TlmForm`** instance (or by a user of **`TlmForm`**). Except for the telemetry packet header, once a packet has been formatted and posted, no further modifications to a telemetry packet are needed. Telemetry buffer management and formatting are treated as two different activities. The **`TlmPkt`** class is used for buffer management, and the **`TlmForm`** subclasses are used for formatting the buffer contents. Only **`TlmPkt`** instances need to be dynamically allocated and released at run-time.

18.4.2 Packet Formats

18.4.2.1 Packet Header

The ACIS produces telemetry packets asynchronously to the spacecraft telemetry frames, and these packets can appear anywhere within the telemetry frame science data sections allocated for ACIS by the spacecraft. Each packet consists of a series of 32-bit words. In order to allow the ground software to easily locate the start of an ACIS telemetry packet within the science stream, ACIS provides a synchronization word at the start of each packet. Gaps between ACIS packets are filled using a hardware generated fill pattern (see

Section 9.4). Within the instrument software, the format of this header is managed by the **TlmPkt** class. Table 21 illustrates the format of the ACIS telemetry packet header.

TABLE 21. Telemetry Packet Header

Word	Bits	Field	Description	Min. Value	Max Value
0	0:31	Packet Synch.	This field contains a constant 32-bit synch. pattern, used to locate the start of a telemetry packet within a spacecraft minor frame.	0x4329da2c	N/A
1	0:9	Packet Length	This field contains the total number of 32-bit words in the packet, including the Packet Synch.	2	1023
1	10:15	Packet Format Tag	This field identifies the type of data contained in the body of the packet.	0	63
1	16:31	Packet Sequence Number	This field contains an telemetry packet counter, which increments for each packet sent by ACIS	0	65,535
2:Packet Length - 1	-	Data	The remainder of the packet contains format tag-specific data.	-	-

18.4.2.2 Packet Formats

Within the ACIS software, each packet data format is represented by some subclass of **TlmForm**. These subclasses inherit directly from **TlmForm**, or inherit from some intermediate abstract class which provides fields common to a set of different formats.

The formats of each packet type are defined by the Instrument Program and Command List (IP&CL) structure tables (see TBD). These format tables are converted into classes either by hand, or using a conversion program. Each generated class is a subclass of **TlmForm**, and uses the **TlmForm** member functions `putField` and `appendField` to write bit-fields into the telemetry packet buffer. The total number of words in a telemetry packet is calculated by each subclass member function `getWordCount`.

18.4.3 Transfer Turn-around Time

In order to reduce the number and size of gaps between telemetry packets when there are a set of packets ready to be sent, the Telemetry Device states that the time between the end of one telemetry transfer, and the start of the next start take less than 0.5 milliseconds (see Section 9.0). In order to avoid possible long delays due to context switching, the Telemetry Manager performs back-to-back telemetry transfers using the **TlmDevice**'s callback instance. When a telemetry transfer completes, the **TlmDevice**'s interrupt handler is invoked. This handler then invokes the installed callback, which in turn, calls the **TlmManager**'s `serviceDevice`() member function. During this interrupt callback

processing, `serviceDevice`() then obtains the next packet from the telemetry queue, and starts the next transfer. If the total time spent handling a telemetry interrupt using a non-empty telemetry queue is less than 0.5 milliseconds (assuming no higher priority interrupts occur and that interrupts are in an enabled state when telemetry transfer completes), no gaps will occur between adjacent, prepared telemetry packets.

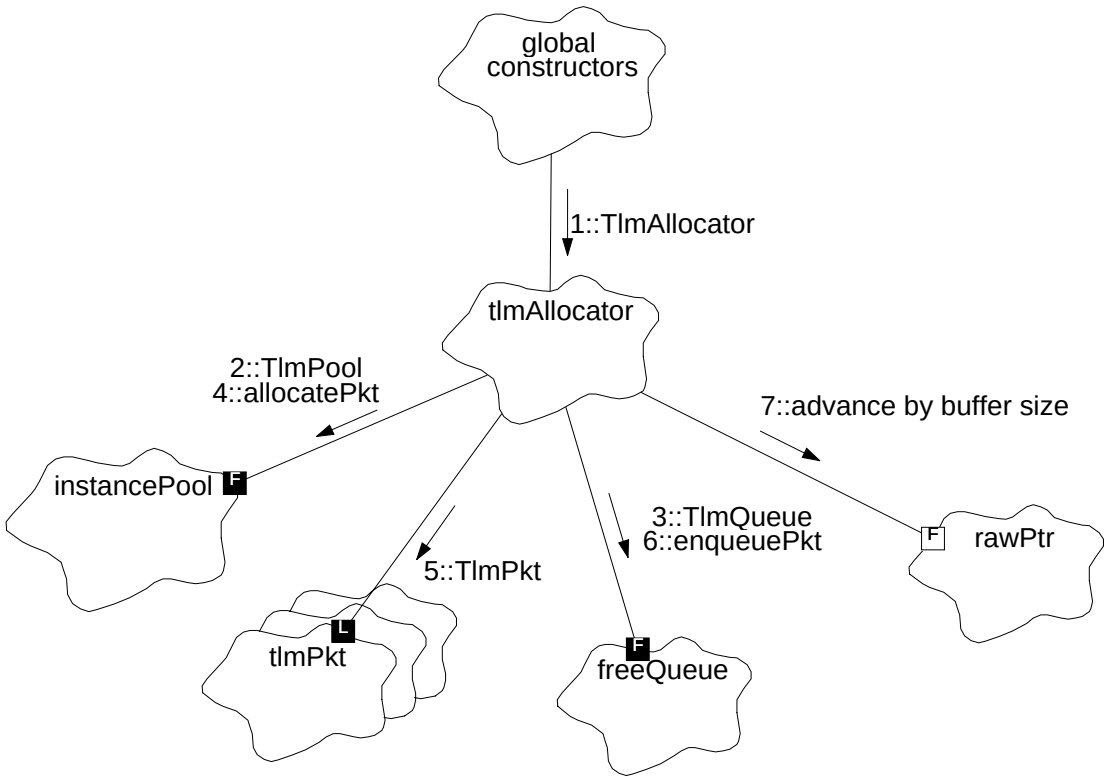
18.5 Scenarios

18.5.1 Use 1:: Allocate telemetry buffers and control structures

Within ACIS, memory for all telemetry buffers and control structures (**`TlmPkt`**) are reserved (via ***`Nucleus RTX`*** memory partitions), allocated and initialized during system initialization and start-up. Once running, groups consisting of equally sized telemetry packet buffers are maintained by **`TlmAllocator`** instances. Any given **`TlmForm`** uses a single **`TlmAllocator`** to obtain packet buffers, and each **`TlmPkt`** has a pointer back to its owning **`TlmAllocator`** instance.

Figure 77 illustrates the allocation of telemetry buffers and control structures during the system initialization process.

FIGURE 77. System Initialization Telemetry Packet Allocation



1. During system start-up, `__main`() is invoked, which calls the constructors for each global **`TlmAllocator`** instance.
2. *`tlmAllocator`* constructor's initialization section calls the constructor for its **`TlmPool`** instance, *`instancePool`*, verifying the size and number of the contained telemetry packet control structure instances. At this point, *`instancePool`* is ready to supply the buffers for all **`TlmPkt`** instances maintained by *`tlmAllocator`*.

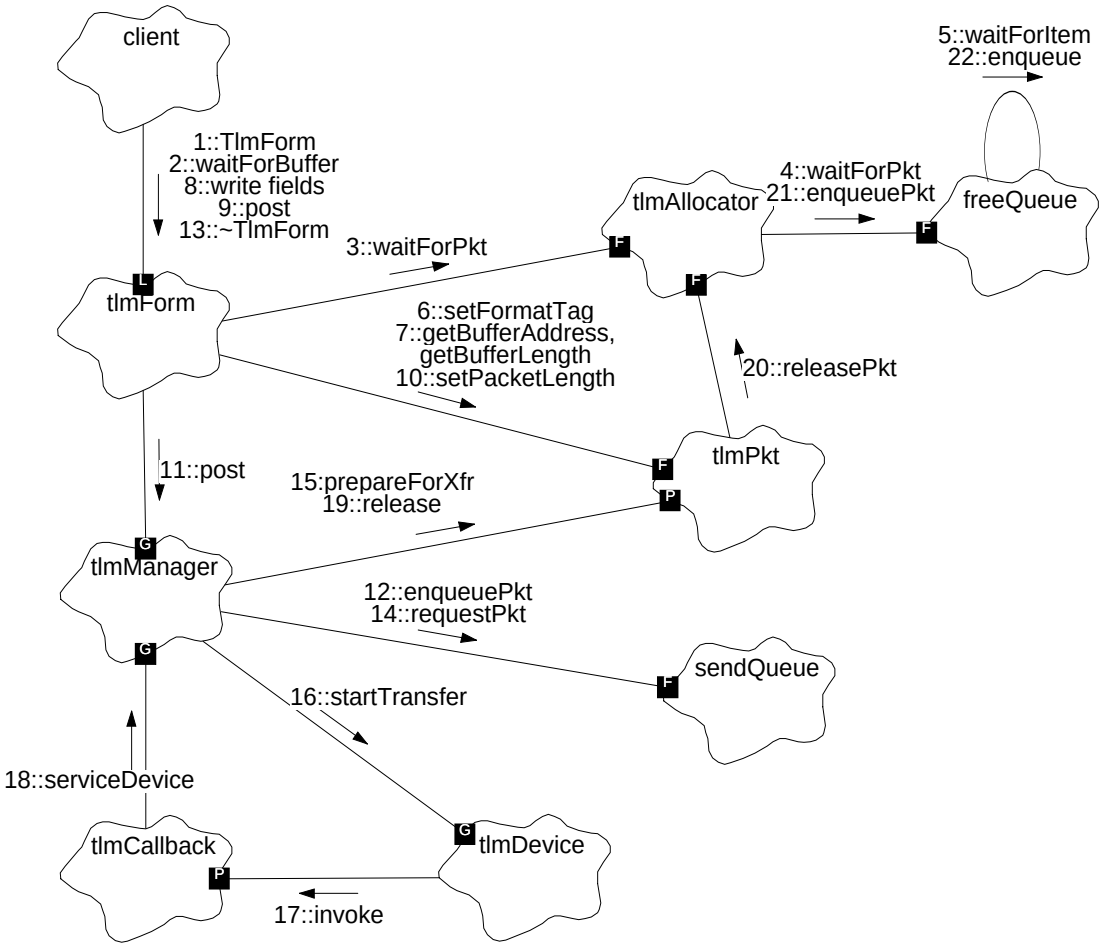
3. *tlmAllocator* constructor's initialization section calls the constructor for its **TlmQueue** instance, *freeQueue*, verifying the maximum number of packet pointers supported by the queue.
4. *tlmAllocator*'s constructor then enters its main body. The body consists of a loop which allocates space for each telemetry packet managed by the allocator instance. The loop starts by calling *instancePool*.allocatePkt() to obtain buffer space to contain a **TlmPkt** instance.
5. The loop then invokes the **TlmPkt** constructor on the obtained buffer, passing *this* as the owner of the packet, and the current value of *rawPtr* as the address of the memory buffer to use for the telemetry information.
6. The loop then invokes *freeQueue*.enqueuePkt() to place the packet on its list of available telemetry packets.
7. Finally, the loop advances *rawPtr* by the size of the telemetry packet buffer.

Once the allocator has been constructed, its *freeQueue* will contain a pointer to each telemetry packet instance maintained by the allocator. Each instance can support a telemetry packet up to the size specified as part to the allocator's constructor.

18.5.2 Use 2:: Manage ordered transfer of telemetry data to hardware

From the client's point of view, all telemetry packet acquisition, formatting and posting is accomplished using subclasses of **TlmForm**. **TlmForm** maintains a pointer to the telemetry packet currently being formatted. Upon construction, this pointer is 0, and the client must invoke **TlmForm**::waitForBuffer() to obtain a telemetry packet. In order to give client code a certain degree of flexibility, this is NOT done automatically by **TlmForm**'s constructor. Once the packet is posted to the telemetry manager, **TlmForm** zeros this pointer to prevent further modifications of the packet's contents, and the client code must call **TlmForm**::waitForBuffer() to obtain another packet. If a **TlmForm** instance is destroyed prior to posting an allocated telemetry packet, its destructor invokes the packet's **TlmPkt**::release() function to ensure that it is not lost. Figure 78 illustrates the overall life-cycle of a **TlmForm** instance, and its formatted telemetry packet.

FIGURE 78. Telemetry formatting, buffering and transfer



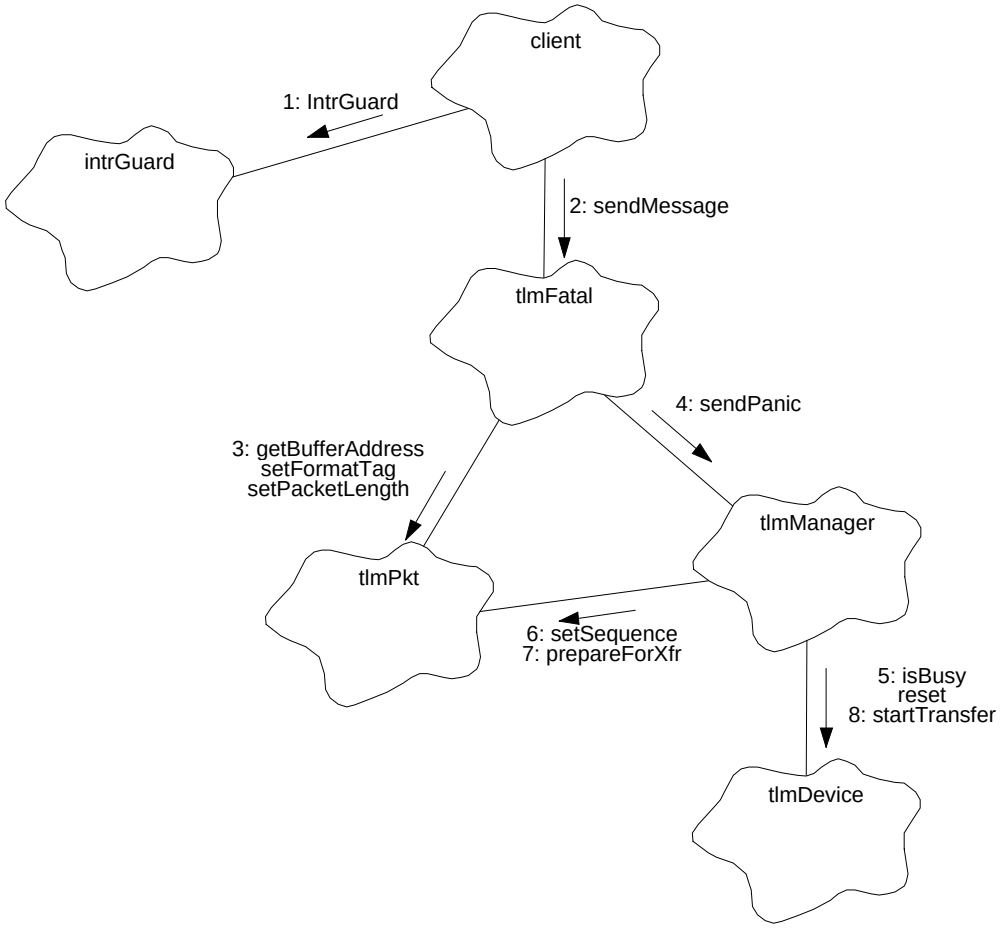
1. A client decides to create and send a telemetry packet. It declares *tlmForm*, whose constructor initializes the state of the instance, and zeros its packet instance pointer.
2. The client then waits for a telemetry packet buffer to become available using *tlmForm*.waitForBuffer().
3. *tlmForm*.waitForBuffer() in-turn calls its allocator's member function, *tlmAllocator*.waitForPkt() to attempt to reserve an unused **TlmPkt** instance.
4. *tlmAllocator*.waitForPkt() then invokes *freeQueue*.waitForPkt().
5. And finally, *freeQueue*.waitForPkt() invokes its protected member function, **Queue**::waitForItem() to block until a telemetry packet pointer becomes available. Once the allocator returns, *tlmForm* retains the acquired packet pointer until it is either posted, (see step 11), or until *tlmForm* is destroyed. If *tlmForm* is destroyed prior to the packet being posted, *tlmForm*'s destructor releases the packet back to its allocator (not shown). If the packet is posted to the *tlmManager*, *tlmForm* is no longer responsible for the packet, and it is *tlmManager*'s responsibility to ensure that the packet is released.

6. Once a packet has been obtained by *tlmForm*, it sets the packet's format tag using *tlmPkt*.setFormatTag().
7. *tlmForm* then obtains and caches the packet's buffer address and length, for use later when the client writes fields into the packet, using *tlmPkt*.getBufferAddress() and *tlmPkt*.getBufferLength().
8. The client then writes zero or more fields into the telemetry packet buffer, using member functions supplied by the format-specific tlmForm instance. The *tlmForm* instance uses the inherited **TlmForm**::putField() and **TlmForm**::appendField() (not shown) functions to write the data into the packet's buffer.
9. Once the client has completed the packet, it tells the *tlmForm* to transfer the packet out of the instrument using *tlmForm*.post().
10. *tlmForm*.post() then computes the number of words written into the packet, using *tlmForm*.getWordCount() (not shown) and passes the result the packet, using *tlmPkt*.setPacketLength().
11. *tlmForm*.post() then invokes *tlmManager*.post(), passing the address of *tlmPkt*.
12. *tlmManager*.post() sets the packet's sequence number using *tlmPkt*.setSequence() (not shown) and then places the packet address on the end of its queue, using *sendQueue*.enqueuePkt(), which then uses **Queue**::enqueue() (not shown).
13. Once *tlmManager*.post() returns, *tlmForm* zeros its local pointer to the packet. This ends *tlmForm*'s responsibility concerning *tlmPkt*, and the client can destroy (~TlmForm) *tlmForm*, without affecting the posted *tlmPkt*.
14. Once the telemetry device is ready, *tlmManager* gets the next telemetry packet to send from the queue using *sendQueue*.requestPkt().
15. *tlmManager* then copies packet header information and obtains the packet's buffer address and length using *tlmPkt*.prepareForXfr().
16. *tlmManager* then tells the telemetry device to transfer the packet, using *tlmDevice*.startTransfer().
17. Once the transfer completes, *tlmDevice*'s interrupt handler invokes the installed telemetry callback, *tlmCallback*.invoke().
18. *tlmCallback*.invoke() then invokes the *tlmManager*.serviceDevice() to release the packet's buffer and start the next transfer (if a packet is available).
19. *tlmManager*.serviceDevice() function then releases the completed packet using *tlmPkt*.release().
20. *tlmPkt*.release() in-turn calls its owner's **TlmAllocator**::releasePkt()
21. *tlmAllocator*.releasePkt() then invokes *freeQueue*.enqueuePkt()
22. Finally, *freeQueue*.enqueuePkt() uses the protected function, **Queue**::enqueue() to place the available packet onto the end of the queue.

18.5.3 Use 3:: Transmission of fatal error messages

The **`TlmManager`** class provides the ability to jam a fatal error message into the telemetry stream. It assumes that, when doing this, all interrupts have been disabled by the client. Figure 79 illustrates the overall fatal error message scenario.

FIGURE 79. Sending a Fatal Error Message



1. The client first ensures that interrupts are disabled by declaring an **`IntrGuard`** instance.
2. The client then issues the fatal error message using *`tlmFatal`*`.sendMessage`().
3. *`tlmFatal`*`.sendMessage` () then uses the various member functions of *`tlmPkt`* to form a fatal error message telemetry packet.
4. The client then sends the message using *`tlmManager`*`.sendPanic`()
5. *`tlmManager`*`.sendPanic`() firsts polls the telemetry device until the current packet being sent completes, using *`tlmDevice`*`.isBusy`(), or until its polling loop counter reaches its limit. It then resets the telemetry hardware using *`tlmDevice`*`.reset`().
6. *`tlmManager`*`.sendPanic`() then sets the sequence number for the fatal message packet using *`tlmPkt`*`.setSequence`().

7. *tlmManager*.sendPanic() then copies the packet's header information into the packet's transfer buffer, and obtains the address and length of the transfer using *tlmPkt*.prepareForXfr().
8. *tlmManager*.sendPanic() then transfers the packet out of the instrument using *tlmDevice*.startTransfer().*tlmManager*.sendPanic() polls the telemetry device until the fatal message packet is sent, using *tlmDevice*.isBusy(), or until its polling loop counter reaches its limit (not shown).

18.6 Class TlmManager

Documentation:

This class manages a queue of telemetry packets, and the transmission of these packets via the instrument telemetry hardware.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Uses:

TlmPkt

Implementation Uses:

TlmDevice *tlmDevice*

Public Interface:

Operations:

```
TlmManager()
post()
sendPanic()
serviceDevice()
```

Private Interface:

Has-A Relationships:

TlmQueue *sendQueue*: This is a queue of pointers to telemetry packets to be sent out of the instrument.

TlmPkt* *curPkt*: This is a pointer to the telemetry packet currently being transferred out of the instrument. If no transfer is underway, this pointer will be 0.

const unsigned *panicTimeout*: This value indicates loop iterations to wait for the telemetry system to go idle, prior to, and after sending a panic message. This value must result in a delay greater than 64 seconds (4K bytes/pkt @ 512 bits/second)

unsigned *curSequence*: This variable contains the sequence number of the next packet to send out of the instrument.

Concurrency: Synchronous

18.6.1 TlmManager()

Public member of: **`TlmManager`**

Arguments:

`unsigned` *`queueId`*
`unsigned` *`maxpkts`*

Documentation:

This constructor initializes the telemetry queue, using the passed ***`Nucleus RTX`*** *`queueId`*. The maximum number of telemetry packets supported by the instrument is specified by *`maxpkts`*. This function also zeros the current transfer packet pointer, indicating that no transfers are underway.

Semantics:

Set *`panicTimeout`* to its initial value, zero *`curPkt`* and *`curSequence`*, and construct *`sendQueue`* using the passed *`queueId`* and with the maximum supported number of packets for the entire system *`maxpkts`*. Then tell the *`tlmDevice`* to install the address of the telemetry manager's interrupt callback instance, *`tlmCallback`*.

Concurrency: Sequential

18.6.2 post()

Public member of: **`TlmManager`**

Return Class: **`void`**

Arguments:

`TlmPkt*` *`pkt`*

Documentation:

This function places the referenced telemetry packet, *`pkt`*, onto the manager's telemetry queue. Once all previously posted packets have been transferred, the referenced packet will be transferred out of the instrument. Once its transfer is complete, its `release`() member function will be invoked.

Semantics:

Set the packet sequence number to *`curSequence`* using *`pkt`*->`setSequence`(). Invoke *`sendQueue`*.`enqueuePkt`() to place the packet on the send queue. If *`curPkt`* is 0, no transfer is underway, then invoke `serviceDevice`() to start a fresh transfer. If *`curPkt`* is not 0, then a packet is transfer in progress. Once the transfer completes, the interrupt callback will start the subsequent transfer.

Concurrency: Synchronous

18.6.3 sendPanic()

Public member of: **`TlmManager`**

Return Class: **`void`**

Arguments:

`TlmPkt*` *`pkt`*

Documentation:

This function “jams” a telemetry packet into the telemetry stream.

Preconditions:

Interrupts must be disabled prior to calling this function

Semantics:

Loop until *`tlmDevice`*`.isBusy()` returns *`BoolFalse`*, or until *`panicTimeout`* is reached. Then reset the telemetry hardware using *`tlmDevice`*`.reset()`. Prepare the packet for transfer and obtain its buffer address and length using *`pkt`*`->prepareForXfr()`. Then tell the telemetry hardware to transfer the packet, *`tlmDevice`*`.startTransfer()`. Once the transfer has started, loop until *`tlmDevice`*`.isBusy()` returns *`BoolFalse`*, or until *`panicTimeout`* is reached.

Concurrency: Sequential

18.6.4 serviceDevice()

Public member of: **`TlmManager`**

Return Class: **`void`**

Arguments:

`IntrDevice*` *`devptr`*

Documentation:

This function is invoked by the telemetry callback instance once a telemetry transfer has been completed. This function invokes the just completed packet's `release`() member function. It then attempts to dequeue another packet from the *`sendQueue`* and start its transfer.

Semantics:

If *`curPkt`* is not 0, then a transfer has just completed. Invoke *`curPkt`*->`release`() to release the packet. Invoke *`sendQueue`*.`requestPkt`() to dequeue a telemetry packet and store the result in *`curPkt`*. If the result is not zero, then store and increment *`curSequence`* into the packet (*`curPkt`*->`setSequence`()), and prepare the packet and obtain its transfer buffer and length using *`curPkt`*->`prepareForXfr`(). Then use *`tlmDevice`*.`startTransfer`() to tell the hardware to transfer the packet.

Concurrency: Synchronous

18.7 Class TlmQueue

Documentation:

This class represents a fixed length queue of telemetry packet pointers.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`Queue`**

Public Uses:

`TlmPkt`

Public Interface:

Operations:
```
TlmQueue()
enqueuePkt()
requestPkt()
waitForPkt()
```

Concurrency: Synchronous

Persistence: Persistent

18.7.1 TlmQueue()

Public member of: **TlmQueue**

Arguments:

unsigned *queueId*
unsigned *nitems*

Documentation:

This constructor initializes the queue of telemetry packet pointers. *queueId* is the ***Nucleus RTX*** queue identifier used for the particular instance being initialized. *nitems* is the number of packet pointers which can be contained in this queue (used for sanity checking against the ***RTX*** queue instance). This function invokes the parent constructor, **Queue**::Queue(), using the number of words in a telemetry packet pointer as the element size within the queue (*queueId* and *nitems* are passed unmodified).

Concurrency: Sequential

18.7.2 enqueuePkt()

Public member of: **TlmQueue**

Return Class: **void**

Arguments:

TlmPkt* *pkt*

Documentation:

This function enqueues a pointer, *pkt*, to a telemetry packet onto this queue, using its parent's **Queue**::enqueue() function.

Concurrency: Synchronous

18.7.3 requestPkt()

Public member of: **TlmQueue**

Return Class: **TlmPkt***

Documentation:

This function attempts to dequeue a packet pointer, using its parent's **Queue**::dequeue() function. If no packets are available, this function returns 0.

Concurrency: Synchronous

18.7.4 waitForPkt()

Public member of: **TlmQueue**

Return Class: **TlmPkt***

Arguments:

unsigned *timeout*

Documentation:

This function waits for and dequeues a packet from the queue, using its parent's **Queue**::waitForItem() function. If no packets are ready, this function waits no longer than *timeout* timer ticks (1/10 second) for one to be enqueued. If *timeout* expires, this function returns 0.

Concurrency: Synchronous

18.8 Class TlmPkt

Documentation:

This class represents a telemetry packet to be transferred out of the instrument.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
TlmPkt()
prepareForXfr()
release()
setFormatTag()
setSequence()
```

Protected Interface:

Operations:
```
getBufferAddress()
getBufferLength()
setPacketLength()
```

Has-A Relationships:

unsigned *pktLength*: This represents the total number of 32-bit words to transfer as part of this telemetry packet. This field is maintained with the packet instance, rather than in the bulk telemetry buffer, due to the single-event upset vulnerability of the main bulk memory. `prepareForXfr`() copies this field to the main buffer just prior to transfer out of the instrument.

unsigned *pktSeq*: This field represents the packet sequence number. It is maintained with the packet instance, rather than with the main bulk memory buffer due to single-event upset considerations. It is copied to the bulk buffer by `prepareForXfr`() just prior to transfer out of the instrument.

`TlmFormatTag` *`pktFormat`*: This is a copy of the packet format tag. It is stored with the packet instance structure separate from the main telemetry buffer due to the vulnerability of the main buffer to single-event upsets. It is copied to the main buffer when `prepareForXfr`() is invoked, just prior to transfer out of the instrument.

Private Interface:

Has-A Relationships:

`unsigned* const` *`bufAddr`*: This is the address in bulk memory allocated for this telemetry packet.

`const unsigned` *`bufLength`*: This is the total number of 32-bit words pointed to by *`bufAddr`* for this telemetry packet.

`const TlmAllocator*` *`owner`*: This is a pointer to the **`TlmAllocator`** instance which allocated the packet. This pointer is used to release the packet once it is no longer needed.

Concurrency: Synchronous

Persistence: Transient

18.8.1 TlmPkt()

Public member of: **TlmPkt**

Arguments:

TlmAllocator* *allocator*
unsigned* *buffer*
unsigned *wordcnt*

Documentation:

This constructor sets up the non-volatile state of a telemetry packet instance, assigning its read-only *owner*, *bufAddr*, and *bufLength* values to the corresponding passed arguments, and zeroing *pktLength*, *pktSeq* and *pktFormat*. *owner* and *buffer* must not be 0, and *wordcnt* must be greater than or equal to 2.

Concurrency: Sequential

18.8.2 getBufferAddress()

Protected member of: **TlmPkt**

Return Class: **unsigned ***

Documentation:

This function returns a pointer to the telemetry packet buffer.

Concurrency: Guarded

18.8.3 getBufferLength()

Protected member of: **`TlmPkt`**

Return Class: **`unsigned`**

Documentation:

This function returns the number of 32-bit words contained in the telemetry packet buffer. The address of the buffer is provided by `getBufferAddress`().

Concurrency: Guarded

18.8.4 setPacketLength()

Protected member of: **`TlmPkt`**

Return Class: **`void`**

Arguments:

`unsigned` *`datacnt`*

Documentation:

This function is used to inform the packet of the amount of information written to its packet buffer. A pointer to the buffer is provided by `getBufferAddress`(). The argument *`datacnt`* reflects the number of 32-bit words written into the entire packet buffer. *`datacnt`* must not exceed the value returned by `getBufferLength`(). This function stores the passed *`datacnt`* into *`pktLength`*.

Concurrency: Guarded

18.8.5 setFormatTag()

Public member of: **`TlmPkt`**

Return Class: **`void`**

Arguments:

`TlmFormatTag` *`tag`*

Documentation:

This function sets the format tag of the telemetry packet instance, by copying *`tag`* into *`pktFormat`*.

Concurrency: Guarded

18.8.6 prepareForXfr()

Public member of: **`TlmPkt`**

Return Class: **`void`**

Arguments:

`unsigned*&` *`xfraddr`*

`unsigned&` *`xfrlen`*

Documentation:

This function prepares a telemetry buffer to be transferred to the telemetry interface hardware. Upon return, *`xfraddr`* will contain the address of the buffer to transfer, and *`xfrlen`* will contain the number of 32-bit words to transfer. NOTE: Given the volatile nature of the telemetry transfer buffer, this function should be called just prior to instructing the telemetry hardware to transfer the buffer.

Semantics:

Copy *`pktLength`*, *`pktFormat`* and *`pktSeq`* into the SEU-soft header space of the buffer, and fill-in the *`xfraddr`* and *`xfrlen`* output arguments.

Concurrency: Synchronous

18.8.7 release()

Public member of: **`TlmPkt`**

Return Class: **`void`**

Documentation:

This function releases a telemetry packet for re-use by invoking *`owner`*`->releasePkt()`.

Concurrency: Synchronous

18.8.8 setSequence()

Public member of: **`TlmPkt`**

Return Class: **`void`**

Arguments:

`unsigned` *`sequence`*

Documentation:

This function sets the sequence number within the packet by copying *`sequence`* to *`pktSeq`*.

Concurrency: Guarded

18.9 Class TlmAllocator

Documentation:

This class is responsible for managing the run-time allocation and releasing of telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

TlmPkt

Public Interface:

Operations:
```
TlmAllocator()
releasePkt()
requestPkt()
waitForPkt()
```

Private Interface:

Has-A Relationships:

TlmQueue *freeQueue*: This queue stores a list of available telemetry packets.

TlmPool *instancePool*: This buffer pool is used gain access to memory reserved for telemetry packet instances. This pool is only used during initialization.

static unsigned* const *rawBase*: This is a shared constant which points to the start of the raw telemetry buffer space in uncached memory.

static const unsigned *rawWordCnt*: This constant shared variable indicates the total number of 32-bit words contained in the raw telemetry buffer space.

static unsigned* *rawPtr*: This is a shared pointer to the next available section of the raw telemetry buffer space, and is used by each **TlmAllocator** instance during setup to reserve a section of the buffer.

18.9.1 TlmAllocator()

Public member of: **`TlmAllocator`**

Arguments:

unsigned *poolId*
unsigned *queueId*
unsigned *wordcnt*
unsigned *npkts*

Documentation:

This constructor initializes its memory pool, *instancePool*, and free packet list, *freeQueue*, using *poolId* and *queueId*, and allocates sections of the raw telemetry buffer, placing references to the allocated packet instances into its *freeQueue*.

Preconditions:

The caller must ensure that no other **`TlmAllocator`** constructor preempts this call. There must be at least (*npkts***wordcnt*) 32-bit words remaining in the raw telemetry buffer space (i.e. *rawPtr* + (*npkts***wordcnt*) <= *rawBase* + *rawWordCnt*).

Semantics:

Initialize *freeQueue* and *instancePool*, passing the number of words in a **`TlmPkt`** instance as the size of each element in the *instancePool*. For each packet (i.e. iterate *npkts* times), allocate a **`TlmPkt`** instance from the *instancePool*, invoke its constructor, and make it available using `releasePkt`(). When constructing the packet, pass the current value of *rawPtr* as the raw telemetry buffer pointer, and advance *rawPtr* by *wordcnt*.

Postconditions:

The *instancePool* will be empty, *rawPtr* will have been advanced by *npkts* wordcnt* words, and *freeQueue* will contain pointers to initialized instances every available telemetry packet associated with this **`TlmAllocator`** instance.

Concurrency: Sequential

18.9.2 releasePkt()

Public member of: **`TlmAllocator`**

Return Class: **`void`**

Arguments:

`TlmPkt*` *`pkt`*

Documentation:

This function releases a telemetry packet back into the *`freeQueue`*, using *`freeQueue`*.`enqueuePkt`().

Concurrency: Synchronous

18.9.3 requestPkt()

Public member of: **`TlmAllocator`**

Return Class: **`TlmPkt*`**

Documentation:

This function attempts to allocate a telemetry packet. If successful, it returns a pointer to the obtained packet. If none are available, it returns 0. This function uses *`freeQueue`*.`requestPkt`() to allocate the next available telemetry packet.

Concurrency: Synchronous

18.9.4 waitForPkt()

Public member of: **`TlmAllocator`**

Return Class: **`TlmPkt*`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function attempts to dequeue a telemetry packet from its *`freeQueue`*. If *`timeout`* expires before a packet becomes available, the function returns 0. If a packet is obtained, it returns a pointer to the constructed packet. This function uses *`freeQueue`*.`waitForPkt`() to wait for and dequeue the packet pointer.

Concurrency: Synchronous

18.10 Class TlmPool

Documentation:

This class represents a pool of buffers used to contain instances of telemetry packets. The intended use of this class is to provide start-up allocation of a patchable number of telemetry packets. As such, this class provides no mechanism for releasing a buffer back into its Memory Pool.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`MemoryPool`**

Public Uses:

`TlmPkt`

Public Interface:

Operations: `TlmPool()`
`allocatePkt()`

Concurrency: Sequential

Persistence: Persistent

18.10.1 TlmPool()

Public member of: **TlmPool**

Arguments:

unsigned *poolId*
unsigned *instanceSize*
unsigned *nbufs*

Documentation:

This constructor initializes a pool of buffers to be used to reserve space for telemetry packet instances. *poolId* is the ***Nucleus RTX*** partition identifier, *instanceSize* and *nbufs* are used to sanity check the **RTX** pool capabilities and are respectively the size of the buffered class instance and number of instances.

Concurrency: Sequential

18.10.2 allocatePkt()

Public member of: **TlmPool**

Return Class: **TlmPkt***

Documentation:

This function retrieves a telemetry packet buffer from the pool. This function uses its parent's **MemoryPool**::allocate() function to acquire a pointer to space reserved for a packet. NOTE: The caller is responsible for invoking the constructor on the returned packet pointer.

Preconditions:

This function must be called no more than once for each reserved telemetry packet instance.

Concurrency: Synchronous

18.11 Class TlmForm

Documentation:

This base class represents a telemetry packet formatter. It is responsible for acquiring a telemetry packet buffer, formatting the contents of the buffer, and for posting the buffer to the telemetry manager. Different types of telemetry packets use different subclasses of this base class.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

`TlmManager` *`tlmManager`*

Public Interface:

Operations:

```
TlmForm()
~TlmForm()
hasBuffer()
post()
requestBuffer()
waitForBuffer()
```

Protected Interface:

Operations:

```
appendField()
getBufLength()
getBufPtr()
getWordCount()
putField()
```

Private Interface:

Has-A Relationships:

`TlmPkt*` *`pktPtr`*: This is a pointer to the current telemetry packet being formatted by this form instance.

`TlmAllocator*` *`allocator`*: This is the telemetry packet buffer allocator used by this instance.

`const TlmFormatTag` *`formatTag`*: This is the telemetry packet format tag to use for this particular type of formatter. It's value is used to set the packet's tag value when the packet is allocated.

`unsigned*` *`bufPtr`*: This is a pointer to the data packet buffer, for use by `putField` and `appendField`.

`unsigned` *`bufLength`*: This is the total number of 32-bit words in the current buffer.

Concurrency: Guarded

Persistence: Transient

18.11.1 TlmForm()

Public member of: **`TlmForm`**

Arguments:

`TlmAllocator*` *`pktsrc`*
`TlmFormatTag` *`tag`*

Documentation:

This constructor initializes the top-level state of the formatter, by setting the read-only allocator instance from *`pktsrc`*, and by zeroing the current packet pointer. *`tag`* is the telemetry packet format tag to use for packets written by this formatter instance.

Postconditions:

`waitForBuffer`(), `requestBuffer`() must be called to obtain a telemetry packet buffer prior to writing to the buffer or posting the packet. The destructor may be invoked without obtaining a buffer.

Concurrency: Guarded

18.11.2 ~TlmForm()

Public member of: **`TlmForm`**

Documentation:

This destructor tests for and releases a telemetry packet buffer, if one has been obtained, but not yet posted.

Concurrency: Guarded

18.11.3 appendField()

Protected member of: **`TlmForm`**

Return Class: **`void`**

Arguments:

`unsigned` *`value`*
`unsigned` *`bitoffset`*
`unsigned` *`bitwidth`*
`unsigned` *`bitmask`*
`unsigned` *`index`*
`unsigned` *`arraywidth`*

Documentation:

This inline function appends a field into the end of a telemetry packet buffer. This function can potentially be more heavily optimized than `putField`() because it does not need to preserve values stored beyond the current field. This function is intended to be expanded within subclass field writer functions, and be heavily optimized by the compiler. *`value`* contains the item to store into the buffer, *`bitoffset`* is the starting bit position of the field within the telemetry packet. If the field is within a structure, the *`bitoffset`* is the bit-position of the first field within the first array element. *`bitwidth`* is the number of bits within the field. *`bitmask`* is a right-justified mask of the field, where 1's correspond to the bits within the field and 0's correspond to other fields. If the field is within an array, *`index`* is the array element to access, and *`arraywidth`* is the number of bits within 1 array element.

Concurrency: Guarded

18.11.4 getBufLength()

Protected member of: **`TlmForm`**

Return Class: **`unsigned`**

Documentation:

This function returns the total number of words available in the current packet buffer.

Concurrency: Guarded

18.11.5 getBufPtr()

Protected member of: **`TlmForm`**

Return Class: **`unsigned*`**

Documentation:

This function returns a pointer to the current packet buffer memory.

Concurrency: Guarded

18.11.6 getWordCount()

Protected member of: **`TlmForm`**

Return Class: **`unsigned`**

Documentation:

This function returns the total number of words currently stored in the packet's buffer.

Concurrency: Guarded

18.11.7 hasBuffer()

Public member of: **`TlmForm`**

Return Class: **`Boolean`**

Documentation:

This function returns whether or not the formatter has an associated telemetry buffer. It returns *`BoolTrue`* is it already has a buffer, and *`BoolFalse`* if it does not.

Concurrency: Guarded

18.11.8 post()

Public member of: **`TlmForm`**

Return Class: **`void`**

Documentation:

This function posts the obtained telemetry packet buffer to the telemetry manager for transfer out of the instrument.

Preconditions:

`waitForBuffer`(), or `requestBuffer`() must obtain a telemetry packet buffer prior to each call to this function.

Postconditions:

The current telemetry packet is disassociated from the formatter, preventing modifications to the packet after being posted to the telemetry manager. Another call to `waitForBuffer`()/`requestBuffer`() must be made prior to using any of the set or get calls.

Concurrency: Guarded

18.11.9 putField()

Protected member of: **TlmForm**

Return Class: **void**

Arguments:

unsigned *value*
unsigned *bitoffset*
unsigned *bitwidth*
unsigned *bitmask*
unsigned *index*
unsigned *arraywidth*

Documentation:

This inline function writes a field into the telemetry packet buffer. This function is intended to be expanded within subclass field writer functions, and be heavily optimized by the compiler. *value* contains the item to store into the buffer, *bitoffset* is the starting bit position of the field within the telemetry packet. If the field is within a structure, the *bitoffset* is the bit-position of the first field within the first array element. *bitwidth* is the number of bits within the field. *bitmask* is a right-justified mask of the field, where 1's correspond to the bits within the field and 0's correspond to other fields. If the field is within an array, *index* is the array element to access, and *arraywidth* is the number of bits within 1 array element.

Concurrency: Guarded

18.11.10 requestBuffer()

Public member of: **TlmForm**

Return Class: **Boolean**

Documentation:

This function attempts to obtain a telemetry packet from the packet allocator. If successful, this function returns *BoolTrue*. If no packets are available at the time of the call, it returns *BoolFalse*.

Concurrency: Guarded

18.11.11 waitForBuffer()

Public member of: **`TlmForm`**

Return Class: **`Boolean`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function uses its allocator to wait for and allocate a telemetry packet. If timeout is reached, this function returns *`BoolFalse`*. If a packet was obtained, it returns *`BoolTrue`*.

Preconditions:

A packet must not have already been allocated, but not yet posted.

Postconditions:

The formatter is ready to accept functions which query for buffer information, or set fields within the packet.

Concurrency: Guarded

18.12 Class TlmCallback

Documentation:

This class handles interrupt callbacks from the telemetry device.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`DevCallback`**

Implementation Uses:

`TlmManager` *`tlmManager`*

Public Interface:

Operations: `invoke()`

Concurrency: Synchronous

Persistence: Persistent

18.12.1 invoke()

Public member of: **`TlmCallback`**

Return Class: **`void`**

Arguments:

`IntrDevice*` *`devptr`*

Documentation:

This function is called by the telemetry device during its interrupt processing. This function invokes the *`tlmManager`*`.serviceDevice()` function to handle the end of a telemetry packet transfer.

Concurrency: Synchronous

18.13 Class TlmFatal

Documentation:

This class represents the formatter for a Fatal Error Message.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

`TlmManager`

Public Interface:

Operations:
`TlmFatal()`
`sendMessage()`

Private Interface:

Has-A Relationships:

`unsigned* const` *`rawFatalBase`*: This is a pointer to the telemetry data buffer to use for the fatal telemetry message.

`TlmPkt` *`pkt`*: This is the reserved telemetry packet used to build and send the fatal message.

`unsigned` *`rawFatalCnt`*: This is the number of words reserved for the fatal error message.

`Boolean` *`rawFatalInUse`*: This indicates that a fatal error message is in the process of being sent.

Concurrency: Synchronous

Persistence: Transient

18.13.1 TlmFatal()

Public member of: **`TlmFatal`**

Documentation:

This constructor creates a global fatal error message. This constructor obtains exclusive access to the global fatal error message telemetry buffer. If it fails to obtain a packet, all functions provided by this instance perform no operation.

Concurrency: Synchronous

18.13.2 sendMessage()

Public member of: **`TlmFatal`**

Return Class: **`void`**

Arguments:

`FatalCode` *`code`*
`unsigned` *`arg`*

Documentation:

This function stores the passed fatal message code and argument into the fatal message telemetry packet, and instructs the Telemetry Manager to transfer the packet out of the instrument. *`code`* indicates which fatal message is to be sent, and *`arg`* is the optional argument of the message.

Concurrency: Synchronous

19.0 Telemetry Packet Formatting Classes (36-53216 A)

19.1 Purpose

The purpose of the telemetry formatting classes are to format the contents of telemetry packets intended for transfer out of the instrument. This section describes the telemetry format class designs. The details of the telemetry packet formats are shown in Section TBD.

19.2 Uses

Use 1:: Format Startup Message Packets
Use 2:: Format Command Echo Packets
Use 3:: Format Software Housekeeping Packets
Use 4:: Format Memory Dump and Execution Result Packets
Use 5:: Format DEA Housekeeping Packets
Use 6:: Format Science and Bias Map Packets

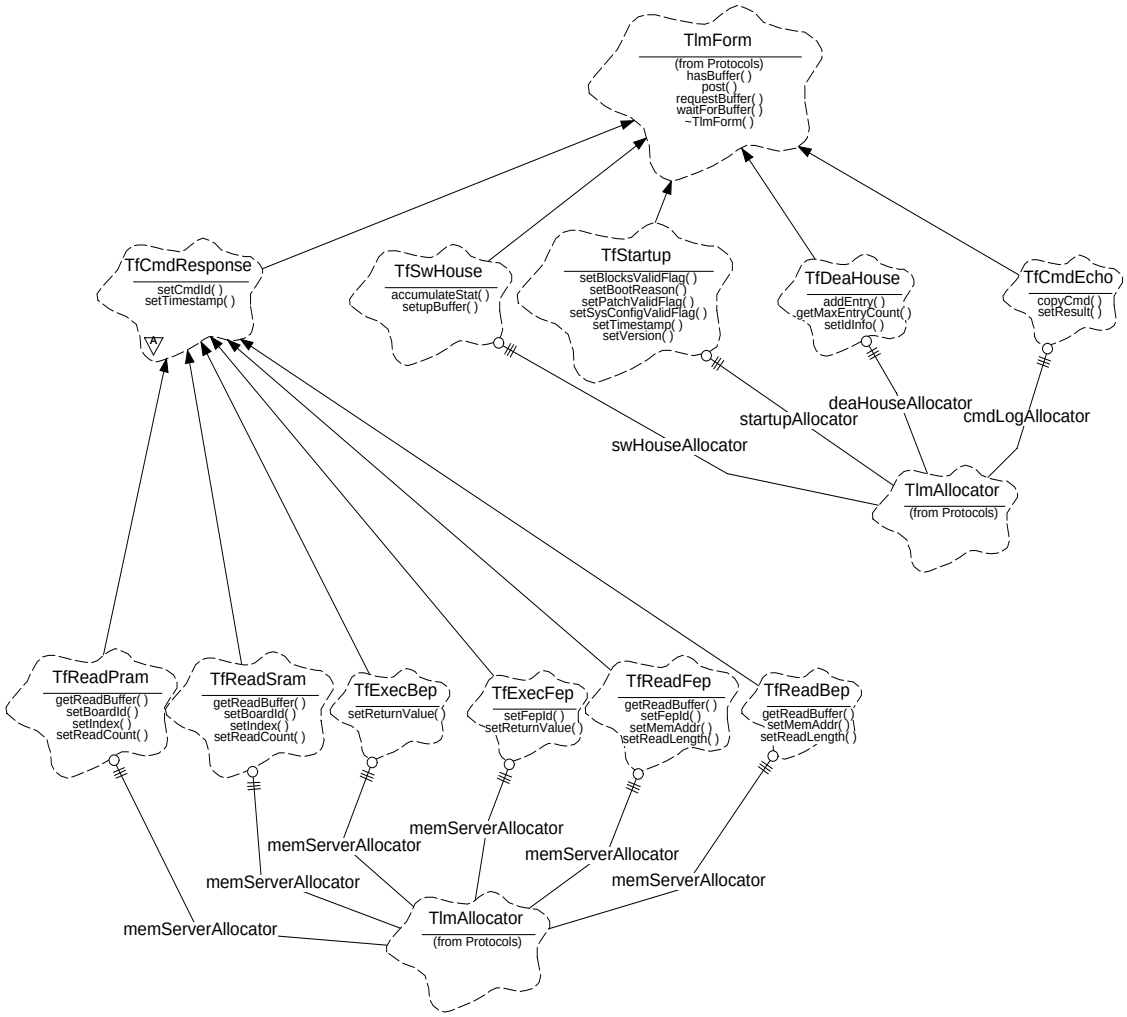
19.3 Organization

All telemetry format classes are a subclass of ***Protocols***::**TlmForm** and packet transfers are managed by the Telemetry Manager (see Section 18.0). Given the large number of telemetry format classes, this section is broken into three subsections: non-science telemetry packet formats, parameter dump formats and science telemetry packet formats.

19.3.1 Non-Science Telemetry Format Relationships

Figure 80 illustrates the overall relationships used by those **TlmForm** subclasses that do not directly format science telemetry. In the diagram, each format class is shown using a **TlmAllocator** instance. The label on each association indicates from which **TlmAllocator** instance a particular format class allocates its telemetry packets (NOTE: Although there are two **TlmAllocator**'s drawn, this is for cosmetic reasons. There is actually only one **TlmAllocator** class definition).

FIGURE 80. Non-Science Telemetry Format Classes



Protocols::**`TlmForm`** - This class is provided by the ***Protocols*** class category, and is responsible for providing the common functionality and interface definitions for all telemetry format classes. See Section 18.0 for a detailed description of this class.

Protocols::**`TlmAllocator`** - This class is provided by the ***Protocols*** class category, and is responsible for allocating and releasing telemetry packet buffers from a telemetry pool. Each pool of telemetry buffers is managed by one allocator instance. After startup, all interactions with instances of this class are made via the **`TlmForm`** member functions. The classes described in this section use the global instances of **`TlmAllocator`** to inform their respective parent **`TlmForm`** which allocator to use to obtain telemetry packets of a particular type. If more than one class uses the same allocator, they compete for buffers from the managed pool. Class instances which use different instances do not compete for buffers (they do, however, compete for telemetry transfer time). See Section 18.0 for more detail.

`TfCmdResponse` - This abstract class is a subclass of **`TlmForm`**, which formats all telemetry packets which are in direct response to a command. This class provides the

common functions to write the command identifier of the initiating command packet, `setCmdId`(), and write the timestamp associated with the response, `setTimestamp`().

`TfSwHouse` - This class is a subclass of **`TlmForm`** and is responsible for accumulating software housekeeping statistics directly into the telemetry packet's buffer, `accumulateStat`(), and for requesting and initializing the telemetry packet buffer, `setupBuffer`(). All instances of this class use the telemetry allocator instance, *`swHouseAllocator`*, to obtain telemetry packet buffers.

`TfStartup` - This class is a subclass of **`TlmForm`** and is responsible for formatting the startup telemetry packet message. This class provides functions to store the software version code, `setVersion`(), (TBD: Patch Version?), the initial ACIS timer tick counter (given an idle telemetry stream, this can be loosely correlated to the telemetry frame number, and subsequently to the spacecraft clock), `setTimestamp`(), the reason for the instrument reset (including the last sent fatal error message code and value), `setBootMode`(), and the results of the initial system integrity checks, `set(Patch,Blocks,SysConfig)ValidFlag()`. All instances of this class obtain their telemetry packet buffers from the *`startupAllocator`* instance.

`TfDeaHouse` - This class is a subclass of **`TlmForm`** and is responsible for formatting the contents of a DEA Housekeeping telemetry packet buffer. This class provides functions to inquire as to the maximum number of entries the buffer can hold, `getMaxEntryCount`(), set the parameter block identifier and timer tick counter into the packet header, `setIdInfo`(), and add housekeeping entries to the body of the packet, `addEntry`(). All instances of this class obtain their telemetry packet buffers from the *`deaHouseAllocator`* instance.

`TfCmdEcho` - This class is a subclass of **`TlmForm`** and is responsible for forming an echo of a command packet in a telemetry packet buffer. This class provides a function to copy a command packet into a command echo telemetry packet buffer, `copyCmd`(), and to set the command execution result code, `setResult`(). All instances of this class obtain their telemetry packet buffers from the *`cmdLogAllocator`* instance.

`TfReadPram` - This class is a subclass of **`TfCmdResponse`**, and is responsible for managing the telemetry packet contents of a dump of one of the DEA CCD Controller's Program RAM (PRAM). This class provides a function to obtain the start of the data buffer to use for the copied PRAM words, and the maximum length of this buffer, `getReadBuffer`(). It also provides functions to store the DEA CCD Controller Board identifier, the starting index of the read, and the number of PRAM words read (`setBoardId`(), `setIndex`() and `setReadCount`() respectively). All instances of this class obtain their telemetry buffers from the *`memServerAllocator`* instance. This **`TlmAllocator`** instance also supplies the following classes with telemetry buffers: **`TfReadSram`**, **`TfExecBep`**, **`TfExecFep`**, **`TfReadBep`**, **`TfReadFep`**.

`TfReadSram` - This class is a subclass of **`TfCmdResponse`**, and is responsible for managing the telemetry packet contents of a dump of one of the DEA CCD Controller's Sequencer RAM (SRAM). This class provides identical capabilities to the **`TfReadPram`**

class, except that its function work using SRAM data units instead of PRAM units. All instances of this class obtain their telemetry buffers from the *memServerAllocator* instance. This **TlmAllocator** instance also supplies the following classes with telemetry buffers: **TfReadPram**, **TfExecBep**, **TfExecFep**, **TfReadBep**, **TfReadFep**.

TfExecBep - This class is a subclass of **TfCmdResponse**, and is responsible formatting the telemetry response to an "Execute Back End Memory" command (see Section TBD). This class provides a function, setReturnValue(), to store the return value of the subroutine called by the execute command. All instances of this class obtain their telemetry buffers from the *memServerAllocator* instance. This **TlmAllocator** instance also supplies the following classes with telemetry buffers: **TfReadSram**, **TfReadPram**, **TfExecFep**, **TfReadBep**, **TfReadFep**.

TfExecFep - This class is a subclass of **TfCmdResponse**, and is responsible for formatting the telemetry response to an "Execute Front End Memory" command (see Section TBD). In addition to providing the setReturnValue() function (see **TfExecBep**), this class also provides a function to store the Front End Processor identifier into the telemetry packet buffer, setFepId(). All instances of this class obtain their telemetry buffers from the *memServerAllocator* instance. This **TlmAllocator** instance also supplies the following classes with telemetry buffers: **TfReadSram**, **TfReadPram**, **TfExecBep**, **TfReadBep**, **TfReadFep**.

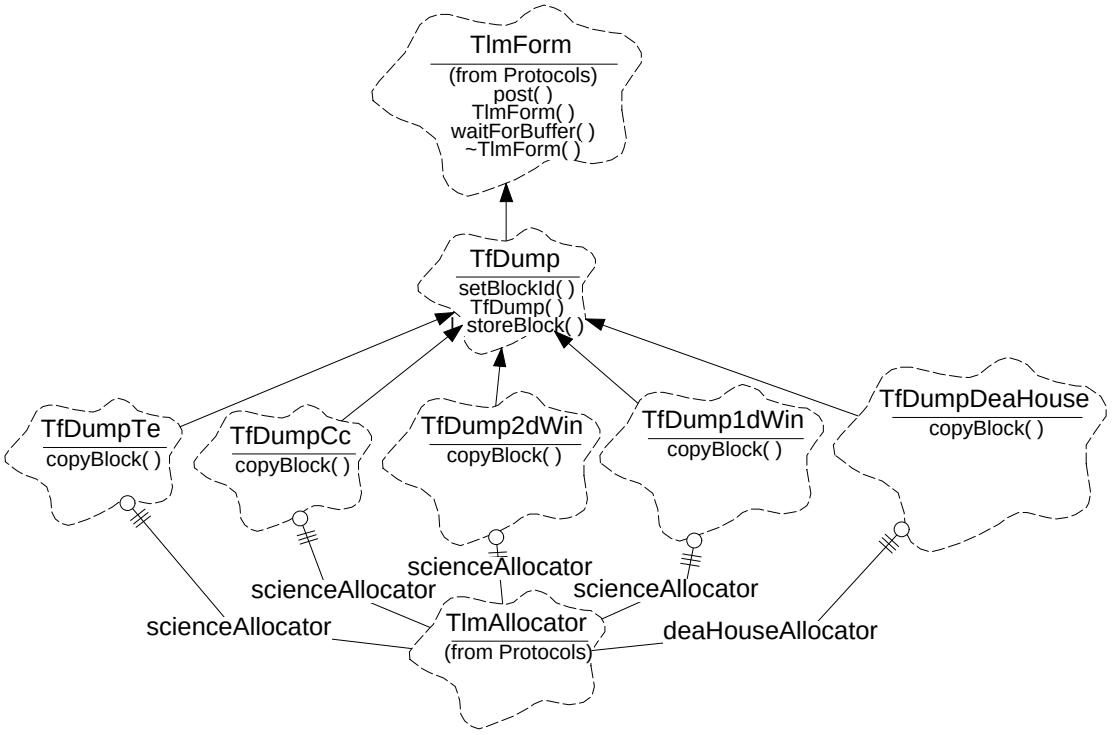
TfReadBep -This class is a subclass of **TfCmdResponse**, and is responsible for managing the telemetry packet contents of a dump of the Back End Processor's memory. This class provides a function to obtain the start of the data buffer to use for the copied words, and the maximum length of this buffer, getReadBuffer(). It also provides functions to store the starting address of the read, and the number of words read (setMemAddr() and setReadCount() respectively). All instances of this class obtain their telemetry buffers from the *memServerAllocator* instance. This **TlmAllocator** instance also supplies the following classes with telemetry buffers: **TfReadSram**, **TfReadPram**, **TfExecBep**, **TfExecFep**, **TfReadFep**.

TfReadFep -This class is a subclass of **TfCmdResponse**, and is responsible for managing the telemetry packet contents of a dump of one of the Front End Processors' memory. This class provides the same set of functions as **TfReadBep** and adds one additional function to store the Front End Processor Identifier, setFepId(). This **TlmAllocator** instance also supplies the following classes with telemetry buffers: **TfReadSram**, **TfReadPram**, **TfExecBep**, **TfExecFep**, **TfReadBep**.

19.3.2 Parameter Dump Telemetry Format Relationships

Figure 81 illustrates the relationships between telemetry packet format classes involved in building parameter blocks during the setup phase of a Science or DEA Housekeeping run (see Sections TBD and TBD). All of the science parameter dumps use the *sienceAllocator* **TlmAllocator** instance to obtain their respective telemetry packet buffers. The DEA Housekeeping parameter dump counts on the *deaHouseAllocator* to supply its telemetry packet buffers.

FIGURE 81. Parameter Dump Telemetry Format Classes



TfDump - This class is a subclass of **TlmForm** and is responsible for storing information in the header area common to all parameter block dump classes. This class provides a function to set the Parameter Block Identifier, `setBlockId`(). It also provides a protected function, used by its subclasses to perform a word-by-word copy of a parameter block into the body of the telemetry packet buffer, `storeBlock`().

TfDumpTe - This class is a subclass of **TfDump** and is responsible for storing a Timed Exposure Parameter block into the body the telemetry packet buffer. This class provides the function `copyBlock`() to perform the copy. This class uses the **TlmAllocator** instance *scienceAllocator*, which is shared will all other science telemetry packet classes except the bias map dump.

TfDumpCc -This class is a subclass of **TfDump** and is responsible for storing a Continuous Clocking Parameter block into the body the telemetry packet buffer. This class provides the function `copyBlock`() to perform the copy. This class uses the

TlmAllocator instance *scienceAllocator*, which is shared will all other science telemetry packet classes except the bias map dump.

TfDump2dWin -This class is a subclass of **TfDump** and is responsible for storing a 2-D Window List Parameter block into the body the telemetry packet buffer. This class provides the function copyBlock() to perform the copy. This class uses the **TlmAllocator** instance *scienceAllocator*, which is shared will all other science telemetry packet classes except the bias map dump.

TfDump1dWin -This class is a subclass of **TfDump** and is responsible for storing a 1-D Window List Parameter block into the body the telemetry packet buffer. This class provides the function copyBlock() to perform the copy. This class uses the **TlmAllocator** instance *scienceAllocator*, which is shared will all other science telemetry packet classes except the bias map dump.

TfDumpDeaHouse -This class is a subclass of **TfDump** and is responsible for storing a DEA Housekeeping Parameter block into the body the telemetry packet buffer. This class provides the function copyBlock() to perform the copy. This class uses the **TlmAllocator** instance *scienceAllocator*, which is shared will all other science telemetry packet classes except the bias map dump.

19.3.3 Science Telemetry Format Relationships

Figure 82 illustrates the relationships between telemetry packet format classes which produce science telemetry. All of these classes, except for the bias map data packets use the *scienceAllocator* **TlmAllocator** instance to obtain their respective telemetry packet buffers. The bias map data uses the *biasAllocator* instance.

Some of the abbreviations used to build the science packet format class names are as follows:

Tf - Telemetry Format (TlmForm)

Sci - Science Format

Exp - Exposure

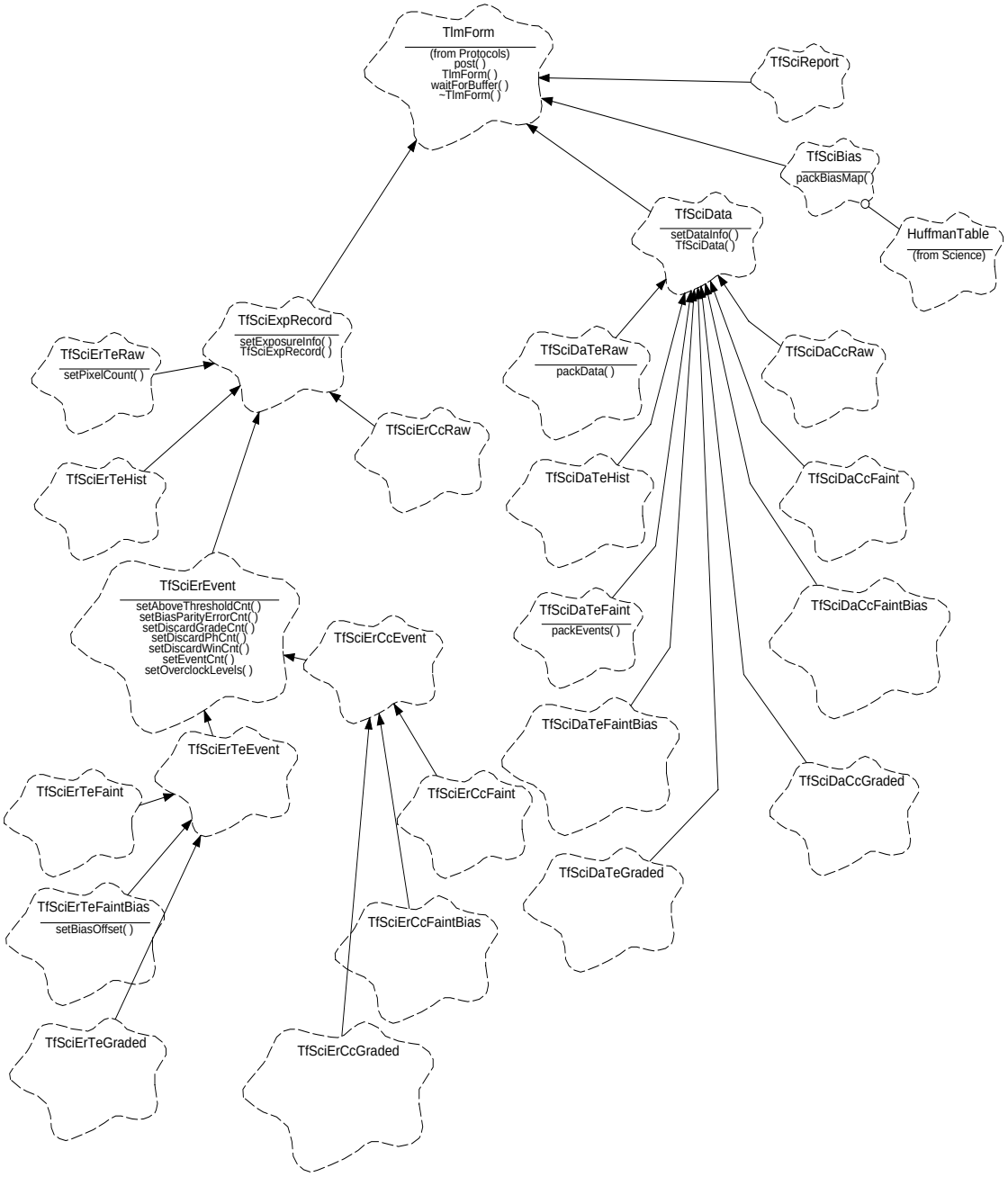
Er - Exposure Record

Da - Data

Te - Timed Exposure

Cc - Continuous Clocking

FIGURE 82. Science Telemetry Format Classes



`TfSciExpRecord` - This class is a subclass of **`TlmForm`** and is responsible for storing information common to all types of Science Exposure Record telemetry packets. This class provides a function to store the parameter block identifier, the microsecond timestamp, latched at the start of the science run, the identifier of the CCD producing the exposure, and the exposure number into the telemetry packet buffer, `setExposureInfo`(). All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

TfSciData - This class is a subclass of **TlmForm** and is responsible for storing information common to all types of Science Data telemetry packets. This class provides a function to store the identifier of the CCD producing the data, and the data packet sequence number for the exposure, setDataInfo(). All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciReport - This class is a subclass of **TlmForm** and is responsible for storing post-science run information, such as a summary of the number of telemetered exposures, the number of bias map parity errors encountered during the run, flags indicating which, if any, of the Front End Processors encountered errors, etc. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciBias - This class is a subclass of **TlmForm** and is responsible for storing bias map information into a telemetry packet buffer. The detailed capabilities of this class are TBD, but will include the ability to pack bias map pixels into the telemetry packet buffer's data area (packBiasMap()). If a compression table is specified, this class uses the passed instance of the **HuffmanTable** class to compress the bias map into the buffer. If no compression table is passed, the bias map is not compressed. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *biasAllocator* (not shown).

TfSciErTeRaw - This class is a subclass of **TfSciExpRecord** and is responsible for forming the body of a Timed Exposure Raw Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD, but will include the ability to store the number of pixels telemetered from the exposure (setPixelCount()). If a compression table is specified, this class uses the passed instance of the **HuffmanTable** class to compress the raw pixels into the buffer. If no compression table is passed, the pixels are not compressed. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciErTeHist - This class is a subclass of **TfSciExpRecord** and is responsible for forming the body of a Timed Exposure Histogram Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciErEvent - This class is a subclass of **TfSciExpRecord** and is responsible for forming the body of all types of event-based Exposure Record telemetry packets. The detailed capabilities of this class are TBD, but will include the ability to store the number of events produced, store the overclock levels used for the exposure, store a count of the number of pixels which exceeded a threshold, store the number of parity errors in the bias map encountered since the start of the run, and store the various event discard counters (see Figure 82). All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

`TfSciErTeEvent` - This class is a subclass of **`TfSciErEvent`** and is responsible for forming the body of all Timed Exposure event-based Exposure Record telemetry packets. The detailed capabilities of this class are TBD.

`TfSciErCcEvent` - This class is a subclass of **`TfSciErEvent`** and is responsible for forming the body of all Continuous Clocking event-based Exposure Record telemetry packets. The detailed capabilities of this class are TBD.

`TfSciErTeFaint` - This class is a subclass of **`TfSciErTeEvent`** and is responsible for forming the body of a Timed Exposure Faint Mode Exposure Record telemetry packet. The detailed capabilities of this class, beyond those of its parent class, are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

`TfSciErTeFaintBias` - This class is a subclass of **`TfSciErTeEvent`** and is responsible for forming the body of a Timed Exposure Faint with Bias Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD, but in addition to the functions provided by its parent, will include the ability to write the Bias Offset level used for the science run into the telemetry packet buffer (`setBiasOffset`()). All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

`TfSciErTeGraded` - This class is a subclass of **`TfSciErTeEvent`** and is responsible for forming the body of a Timed Exposure Graded Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

`TfSciErCcRaw` - This class is a subclass of **`TfSciExpRecord`** and is responsible for forming the body of a Continuous Clocking Raw Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

`TfSciErCcFaint` - This class is a subclass of **`TfSciErCcEvent`** and is responsible for forming the body of a Continuous Clocking Faint Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

`TfSciErCcFaintBias` - This class is a subclass of **`TfSciErCcEvent`** and is responsible for forming the body of a Continuous Clocking Faint Bias Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

TfSciErCcGraded - This class is a subclass of **TfSciErCcEvent** and is responsible for forming the body of a Continuous Clocking Graded Mode Exposure Record telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaTeRaw - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Timed Exposure Raw Mode Data telemetry packet. The detailed capabilities of this class are TBD, but will include the ability to pack raw pixel data into the telemetry packet buffer (packData()). All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaTeHist - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Timed Exposure Histogram Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaTeFaint - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Timed Exposure Faint Mode Data telemetry packet. The detailed capabilities of this class are TBD, but will include the ability to pack Faint-Mode events and bias map parity error indicators into the telemetry packet buffer (packEvents()). All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaTeFaintBias - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Timed Exposure Faint with Bias Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaTeGraded - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Timed Exposure Graded Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaCcRaw - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Continuous Clocking Raw Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

TfSciDaCcFaint - This class is a subclass of **TfSciData** and is responsible for forming the contents of a Continuous Clocking Faint Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **TlmAllocator** instance, *scienceAllocator* (not shown).

`TfSciDaCcFaintBias` - This class is a subclass of **`TfSciData`** and is responsible for forming the contents of a Continuous Clocking Faint with Bias Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

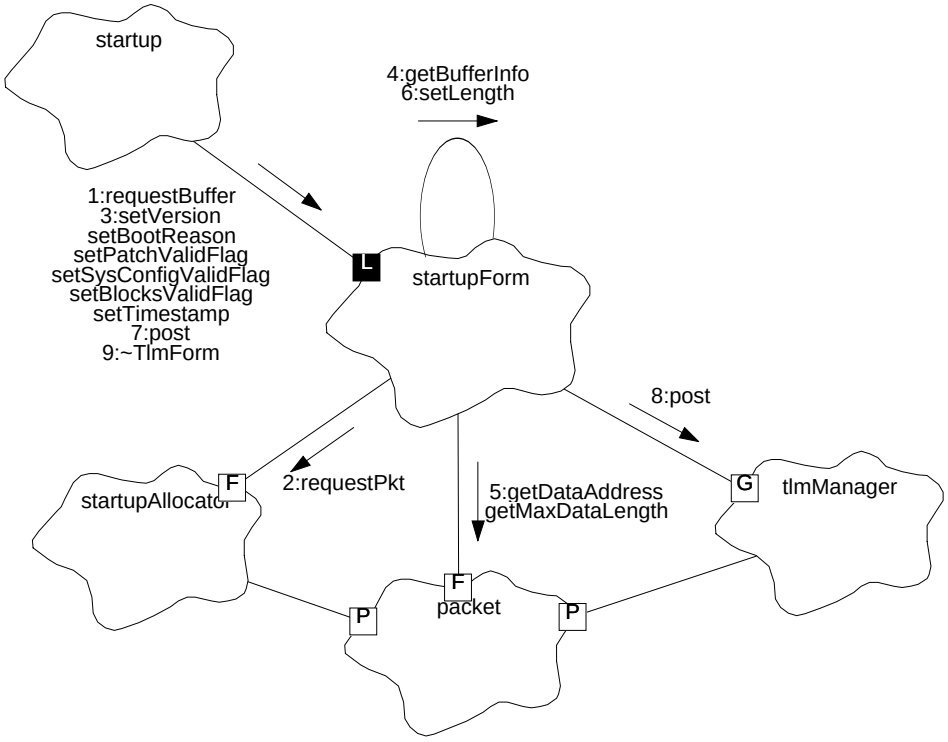
`TfSciDaCcGraded` - This class is a subclass of **`TfSciData`** and is responsible for forming the contents of a Continuous Clocking Graded Mode Data telemetry packet. The detailed capabilities of this class are TBD. All instances of this class obtain their telemetry packet buffers from the **`TlmAllocator`** instance, *`scienceAllocator`* (not shown).

19.4 Scenarios

19.4.1 Use 1: Format Startup Message Packets

Figure 83 illustrates the sequence of events involved in forming and sending the startup telemetry packet. This scenario re-iterates some of the behaviors described in Section 18.0.

FIGURE 83. Forming and sending a Startup Message



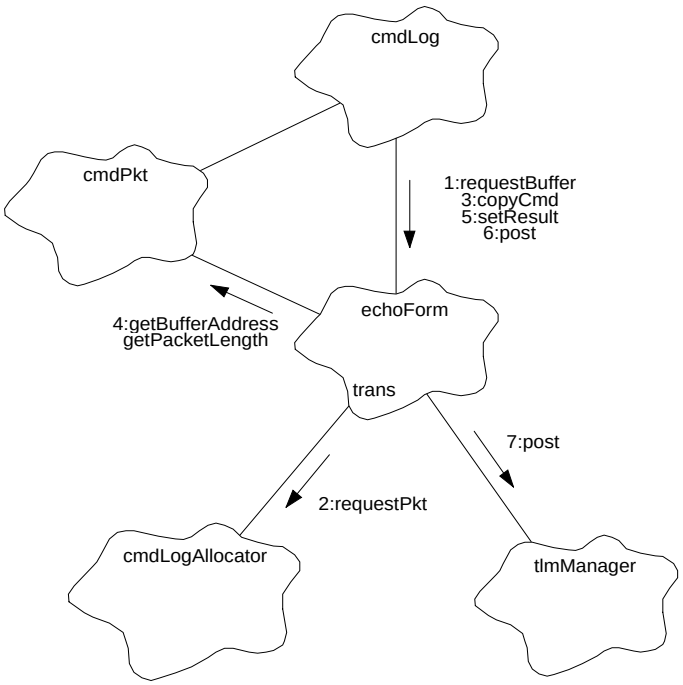
1. The *startup* unit initializes the ACIS software, performs initial integrity checks of all persistent data structures, starts the executive, constructs all global class instances, and constructs a local **TfStartup** instance, *startupForm* (steps not shown). It then asks the form to obtain a telemetry packet buffer from its pool, using *startupForm*.requestBuffer().
2. *startupForm*.requestBuffer(), inherited from **TlmForm**, invokes its allocator's *allocator*->requestPkt() function, which returns a **TlmPkt** instance, *packet*.
3. *startup* tells *startupForm* to set the software version code, the reason for the re-boot, the various results of its earlier integrity checks, and the value of the Back End's timer tick counter (setVersion(), setBootReason(), setPatchValidFlag(), setSysConfigValidFlag(), setBlocksValidFlag(), setTimestamp(), respectively).
4. When asked to store these data elements, *startupForm* uses **TlmForm**::getBufferInfo() to obtain the address and length of the data portion of the packet. It then stores the value into the packet's data buffer.

5. When invoked, **`TlmForm`**::`getBufferInfo`() uses the packet's `getDataAddress`() and `getMaxDataLength`() functions to obtain the address and length of the telemetry packet's data buffer.
6. Once a value is stored, ***`startupForm`*** uses **`TlmForm`**::`setLength`() to set the total packet length. **`TlmForm`**::`setLength`() uses the packet's `setDataLength`() to record the total length of the data portion of the telemetry packet (not shown).
7. Once all of the fields have been set, ***`startup`*** tells ***`startupForm`*** to post the packet for transfer, using **`TlmForm`**::`post`().
8. **`TlmForm`**::`post`() in-turn, passes a pointer to the packet to the telemetry manager, using ***`tlmManager`***.`post`(). At this point, the packet is placed into the telemetry queue, awaiting transfer out of the instrument. Once the packet has been transferred, the telemetry manager releases the packet back to its allocator, ***`startupAllocator`*** (sequence not shown).
9. Once the packet has been posted, ***`startup`*** is free to release the form (~`TlmForm`) (since it is a local, this is done by leaving the scope which declared the instance).

19.4.2 Use 2: Format Command Echo Packets

Figure 84 illustrate the overall sequences used to echo command packets.

FIGURE 84. Command Echo formatting

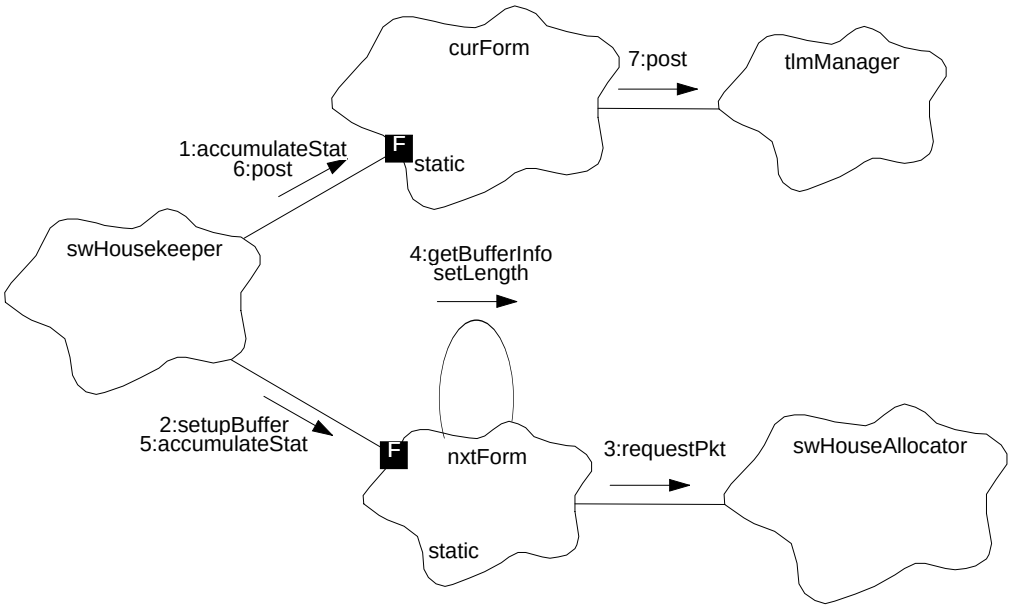


1. The *cmdLog* (see Section TBD) is asked to open an entry. It asks its local *echoForm* to obtain a telemetry buffer, using *echoForm*.requestBuffer() (inherited from **TlmForm**).
2. *echoForm*.requestBuffer() then asks *cmdLogAllocator* for a telemetry packet buffer, using the non-blocking *allocator*->requestPkt(). If the call returns a buffer, the scenario proceeds, otherwise, a dropped echo is reported to software housekeeping (not shown), and the *cmdLog* aborts its attempt to echo the command.
3. Once a packet has been obtained, *cmdLog* instructs *echoForm* to copy the command packet into the telemetry packet buffer, using *echoForm*.copyCmd(), passing a pointer to the packet to be echoed.
4. copyCmd() calls *cmdPkt*.getBufferAddress() and *cmdPkt*.getPacketLength() to get the pointer and total length of the command packet to be echoed. It then uses **TlmForm**::getBufferInfo() to get the destination address and length (not shown), and copies the command packet contents into the telemetry buffer. It then calls **TlmForm**::setLength() to inform the telemetry packet of the amount of data copied (also not shown).
5. Later, after the command has been executed, the *cmdLog* is told to close the entry. It stores the passed command result code into the telemetry packet buffer using *echoForm*.setResult().
6. *cmdLog* tells *echoForm* to post the formed telemetry packet to telemetry, using the inherited **TlmForm**::post()
7. **TlmForm**::post() then passes a pointer to the telemetry packet buffer to *tlmManager*.post() to be queued for transfer out of the instrument, and zeros the local packet pointer.

19.4.3 Use 3: Format Software Housekeeping Packets

Figure 85 illustrates the overall scenario for logging statistics into software housekeeping packet buffers, and the steps used to post a housekeeping packet for transfer out of the instrument. This scheme relies on the client, the Software Housekeeper, maintaining two instances of **`TfSwHouse`**, referenced via pointers. When a transfer is requested, one form is used to acquire a telemetry buffer, and setup the buffer, while the other form is used to continue to accumulate asynchronous statistics reports. Once the new buffer is setup, the client starts acquiring values into the new buffer, while the second is posted for transfer out of the instrument.

FIGURE 85. Software Housekeeping format and posting



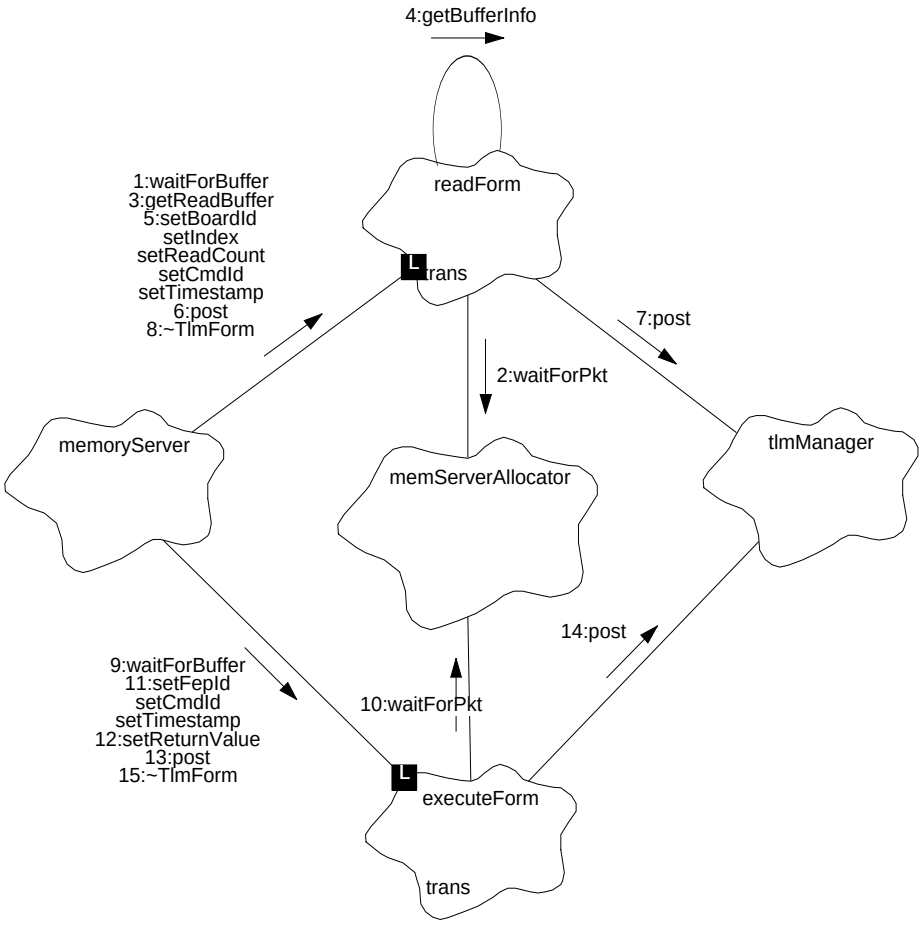
1. Initially, *`curForm`* has a telemetry buffer, and *`nxtForm`* does not. All housekeeping reports are forwarded by *`swHousekeeper`* to *`curForm`*, which increments the corresponding statistic slot and stores the associated value directly into the telemetry buffer, using c*`urform`*->`accumulateStat`().
2. When the *`swHousekeeper`* wants to send the current housekeeping log, it invokes *`nxtForm`*->`setupBuffer`() to attempt to obtain a telemetry packet buffer.
3. `setupBuffer`() calls its inherited function, **`TlmForm`**::`requestBuffer`() (not shown), which then calls the *`swHouseAllocator`* to attempt to obtain a buffer using, *`allocator`*->`requestPkt`(). If no buffers are available, `setupBuffer`() fails, and the housekeeper reports the occurrence to *`curForm`*, and the scenario stops.
4. If a buffer is acquired, *`nxtForm`* calls its inherited function, **`TlmForm`**::`getBufferInfo`() to obtain the address and space available for use for the statistics. It then calls **`TlmForm`**::`setLength`() to establish the length of the housekeeping header and statistics entry array portion of the packet.

5. Once *nxtForm* is setup, *swHousekeeper* uses the form to accumulate any new statistics, *nxtForm*->accumulateStat().
6. At this point, new statistics reports are no longer being sent to *curForm*, and the *swHousekeeper* tells *curForm* to send its telemetry packet, using **TlmForm**::post().
7. **TlmForm**::post() then passes the telemetry packet pointer to *tlmManager*.post() to be queued for transfer out of the instrument. **TlmForm**::post() then zeros *curForm*'s telemetry packet pointer. At this point, *nxtForm* has a telemetry buffer, and *curForm* does not, and all software housekeeping reports are accumulated into *nxtForm*'s telemetry packet buffer.

19.4.4 Use 4: Format Memory Dump and Execution Result Packets

Figure 86 illustrates the use of the memory dump and execution result telemetry formats. The diagram illustrates the use of the **TfReadPram** and **TfExecFep** classes. The use of the remaining memory server classes, **TfReadBep**, **TfReadFep**, **TfReadSram**, **TfExecBep**, are similar and are not explicitly described.

FIGURE 86. Memory Dump and Execution Form Use



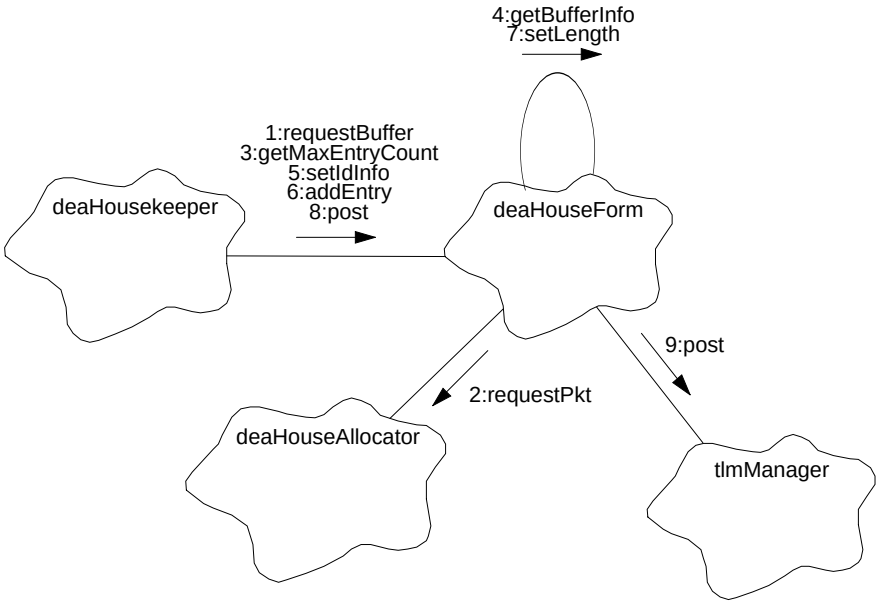
1. The *memoryServer* receives a request to read and telemeter a portion of one of the CCD Controller's PRAM and enters a loop which terminates once the entire request has been satisfied. At the top of the loop, the *memoryServer* constructs an instance of **TfReadPram**, called *readForm*, and calls *readForm*.waitForBuffer().
2. *readForm*.waitForBuffer() goes to *memServerAllocator* to obtain a buffer, using *allocator*->waitForPkt().
3. The *memoryServer* then calls *readForm*.getReadBuffer() to obtain a pointer to an array of PRAM data slots, and a count of the maximum number of data words the packet can hold.
4. *readForm*.getReadBuffer() uses **TfCmdResponse**::getBufferInfo() to obtain the data area of the packet (after packet header and command response headers).
5. Once the *memoryServer* has the destination buffer, it acquires and fills the buffer with PRAM data words. The *memoryServer* then stores the packet's header information, by calling *readForm*'s setBoardId(), setIndex(), setCmdId(), and setTimestamp(). It informs the packet how many PRAM words were actually written, using *readForm*.setReadCount(). Except for setReadCount(), each of these **TfReadPram** functions write their respective data directly into the telemetry buffer. setReadCount(), however, adds in its own header size to the reported value, and then uses **TfCmdResponse**::setLength() (not shown) to establish the length of its portion of the packet.
6. The *memoryServer* then invokes *readForm*'s inherited **TlmForm**::post() to transfer packet information out of the instrument
7. **TlmForm**::post() passes the telemetry packet pointer to *tlmManger*.post(), and zeros the local copy.
8. The *memoryServer* then invokes the destructor (~TlmForm()) as it iterates or exits its loop.
9. Later, the *memoryServer* is asked to execute a subroutine in one of the Front End Processor memories, and telemeter the result. The *memoryServer* declares an instance of **TfExecFep** called *executeForm*, and calls its waitForBuffer() function, inherited from **TlmForm**.
10. **TlmForm**::waitForBuffer() invokes the *memServerAllocator* to get a buffer, using *allocator*->waitForPkt().
11. The *memoryServer* then invokes *executeForm*'s setFepId(), setCmdId(), and setTimestamp(), to write the specified information into the telemetry buffer.
12. The *memoryServer* executes the specified subroutine on the indicated Front End Processor, and retrieves the value returned by the called function (not shown). It then stores the return value into the telemetry packet buffer using *executeForm*.setReturnValue().
13. The *memoryServer* posts the telemetry packet for transfer using *executeForm*.post() (inherited from **TlmForm**).

14. **`TlmForm`**::`post`() passes *`tlmManager`*.`post`() the pointer to the telemetry packet, and zeros its local copy.
15. The *`memoryServer`* then destroys *`executeForm`* (~`TlmForm`()) by leaving the scope which declared the form.

19.4.5 Use 5: Format DEA Housekeeping Packets

The DEA Housekeeper periodically acquires and telemeters a set of DEA housekeeping values. Figure 87 illustrates the use of DEA Housekeeping formats to accumulate and send one set of DEA Housekeeping information.

FIGURE 87. DEA Housekeeping format and posting



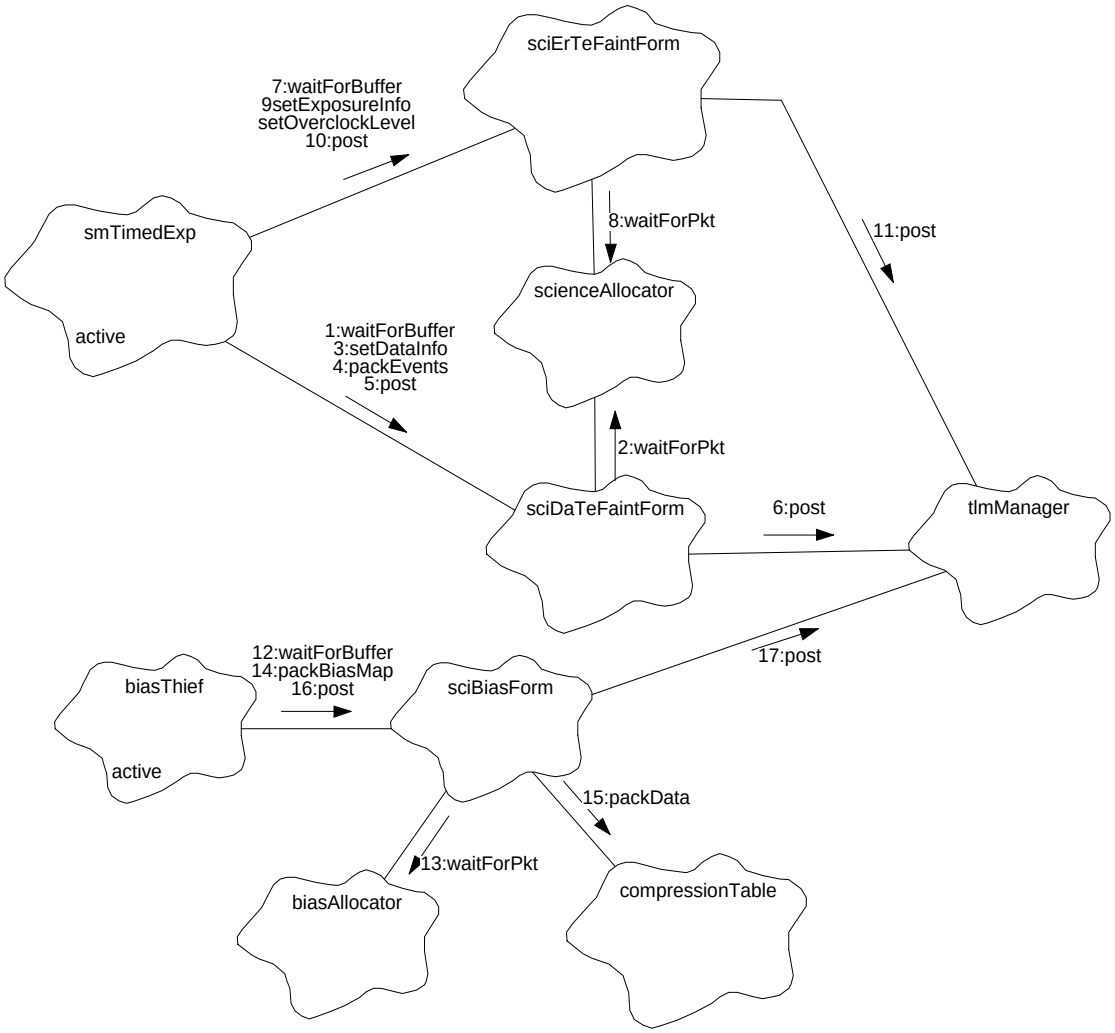
1. The *`deaHousekeeper`* starts its data acquisition, constructs a fresh instance of a **`TfDeaHouse`** form, called *`deaHouseForm`*, and asks it to request a telemetry buffer, using *`deaHouseForm`*.`requestBuffer`(). If it fails, the housekeeper logs the occurrence in software housekeeping (not shown) and tries again later, at the start of its next phase.
2. *`deaHouseForm`*.`requestBuffer`() uses *`deaHouseAllocator`*.`requestPkt`() to acquire its telemetry packet buffer.
3. The *`deaHousekeeper`* queries the form for its maximum number of supported entries, using *`deaHouseForm`*.`getMaxEntryCount`().
4. `getMaxEntryCount`() uses the inherited function **`TlmForm`**::`getBufferInfo`() to determine the amount of space available in the telemetry packet for housekeeping entries (minus any header space used by **`TfDeaHouse`** itself).

5. *deaHousekeeper* stores the parameter block identifier used for the run, and the current Back End Processor timer tick counter, into the telemetry packet buffer using *deaHouseForm*.setIdInfo().
6. *deaHousekeeper* then queries the DEA subsystems for a piece of housekeeping information specified by the selected DEA Housekeeping Parameter block, and appends it data in the telemetry packet buffer using *deaHouseForm*.addEntry(). It continues to do this for each housekeeping entry in the DEA Housekeeping Parameter block, or until the maximum number of supported entries is reached in the telemetry packet.
7. As each entry is added to the telemetry packet, *deaHouseForm* informs the packet of the added data using the inherited **TlmForm**::setLength().
8. Once all of the housekeeping values have been acquired and stored, the *deaHousekeeper* calls *deaHouseForm*.post() to post the packet for transfer out of the instrument.
9. post() then passes *tlmManager*.post() the pointer to the telemetry packet, and zeros its local copy.

19.4.6 Use 6: Format Science Packets

Figure 88 illustrates prototype uses of the Science Telemetry forms. The scenario illustrates the use of Timed Exposure Faint Mode exposures and data, concurrently with the sending of the bias map. Steps 1 - 11 illustrate the overall scenario for exposure data from one CCD. Steps 12 - 17 illustrate the steps used by the Bias Thief to form and send Bias Map telemetry packets.

FIGURE 88. Timed Exposure Faint Mode telemetry formats with Bias Map



1. The current science mode, *`smTimedExp`*, receives a new exposure's worth of Faint Mode events from one of the Front End's processing CCD images. *`smTimedExp`* constructs a science data timed exposure faint mode form (**`TfSciDaTeFaint`**) instance, *`sciDaTeFaintForm`*, and asks the form to acquire a telemetry packet buffer, using *`sciDaTeFaintForm`*.`waitForBuffer`().
2. `waitForBuffer`() calls *`scienceAllocator`*.`waitForPkt`() to wait for and allocate a telemetry packet buffer. If `waitForPkt`()'s time-out expires, the science mode performs any needed housekeeping operations, and then retries until one becomes available.
3. Once a buffer has been allocated, *`smTimedExp`* stores the CCD identifier and data packet sequence number into the telemetry buffer using *`sciDaTeFaintForm`*.`setDataInfo`().

4. *smTimedExp* then calls *sciDaTeFaintForm*.packEvents()to pack the received events into the telemetry buffer. As new events from the CCD exposure arrive, the mode continues to call *sciDaTeFaintForm*.packEvents() until the telemetry buffer fills.
5. Once the telemetry buffer becomes full, or once all of the exposure data from the CCD have been packed, *smTimedExp* calls the inherited function **TlmForm**::post()) to transfer the packet to telemetry.
6. **TlmForm**::post() passes *tlmManager*.post() the pointer to the telemetry buffer, which then queues the packet for transfer out of the instrument. **TlmForm**::post() then zeros the local copy of the telemetry packet pointer, and the form can be destroyed. If there are more events from the CCD exposure to process, *smTimedExp* constructs a new form, and asks the form to wait for a new buffer. Once a buffer has been obtained, the scenario proceeds from step 3. (NOTE: The construction/destruction of the form is not completely necessary and is left as an option to the mode).
7. Once all events from a CCD exposure have been packed and posted to telemetry, *smTimedExp* constructs a Timed Exposure Faint Mode Record form, *sciErTeFaintForm*, and asks the form to wait for and allocate a telemetry buffer, *sciErTeFaintForm*.waitForBuffer().
8. As in step 2, waitForBuffer() calls *scienceAllocator*.waitForPkt() to wait for and allocate a telemetry packet buffer. If the call times-out, the science mode performs any needed housekeeping operations, and then retries until one becomes available.
9. Once a buffer has been allocated, *smTimedExp* stores the CCD identifier, science run starting timestamp, the parameter block identifier, and the exposure number into the telemetry packet buffer using *sciErTeFaintForm*.setExposureInfo(). It also stores the overclock levels used by the completed exposure, using *sciErTeFaintForm*.setOverclockLevel().
10. *smTimedExp* then sends the packet using the inherited function, **TlmForm**::post().
11. As in step 6, **TlmForm**::post() invokes *tlmManager*.post() and zeros its local telemetry packet pointer.
12. Meanwhile, the *biasThief* is attempting to telemeter the bias maps being used by each of the active Front End Processors. As science processing periodically yields control to the *biasThief*, the *biasThief* constructs a **TfSciBias** instance, *sciBiasForm*, and asks it to wait for a telemetry buffer from its own pool, using *sciBiasForm*.waitForBuffer() (inherited from **TlmForm**).
13. waitForBuffer() attempts to obtain a buffer using the bias thief's packet allocator, *biasAllocator*.waitForPkt(). If the wait times out, the bias thief performs any needed housekeeping, and then retries until it successfully obtains a buffer.
14. Once *sciBiasForm* gets a telemetry packet buffer, the *biasThief* calls *sciBiasForm*.packBiasMap(), passing a reference to the region of the FEP's bias map to send (including row/column information), and a pointer to the Huffman Compression table to use, *compressionTable*.

15. *sciBiasForm*.packBiasMap() then uses the passed table to compress the pixel bias directly from the FEP's shared-memory bias map area, into the telemetry packet buffer, using *compressionTable*->packData(). (NOTE: If the passed *compressionTable* pointer is 0, *sciBiasForm* just packs the data into the telemetry buffer without compression.
16. Once the telemetry packet buffer is full, the *biasThief* posts the packet for transfer, using *sciBiasForm*.post().
17. *sciBiasForm*.post() then passes the packet buffer pointer to *tlmManager*.post() for transfer out of the instrument. *sciBiasForm*.post() then zeros its local telemetry packet pointer. The *biasThief* then repeats from step 12 until the remaining bias map information is packed and sent.

19.5 Class TfCmdEcho

Documentation:

This class formats command echo telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Uses:

`CmdPkt`

Implementation Uses:

`TlmAllocator` *`cmdLogAllocator`*

Public Interface:

Operations:
```
copyCmd()
setResult()
```

Concurrency: Guarded

Persistence: Transient

19.5.1 copyCmd()

Public member of: **TfCmdEcho**

Return Class: **void**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function copies the contents of the passed command packet, *cmdpkt*, into the telemetry buffer.

Concurrency: Guarded

19.5.2 setResult()

Public member of: **TfCmdEcho**

Return Class: **void**

Arguments:

CmdResult *result*

Documentation:

This function copies the command execution result code, *result*, into the telemetry buffer.

Concurrency: Guarded

19.6 Class TfCmdResponse

Documentation:

This is an abstract class which is responsible for formatting the command identifier portion of all telemetry packets which are in direct response to a command.

Export Control: Public

Cardinality: 0

Hierarchy:

Superclasses: **`TlmForm`**

Public Interface:

Operations:
```
TfCmdResponse()
setCmdId()
setTimestamp()
```

Protected Interface:

Operations:
```
getBufferInfo()
setLength()
```

Concurrency: Guarded

Persistence: Transient

19.6.1 TfCmdResponse()

Public member of: **TfCmdResponse**

Arguments:

TlmAllocator* *allocator*
TlmFormatTag *format*

Documentation:

This constructor initializes its parent, **TlmForm**, passing up a pointer to packet buffer manager, *allocator*, and the telemetry format tag to use for this instance, *format*.

Concurrency: Guarded

19.6.2 getBufferInfo()

Protected member of: **TfCmdResponse**

Return Class: **void**

Arguments:

unsigned*& *bufaddr*
unsigned& *bufcnt*

Documentation:

This function supplies the buffer address, written into *bufaddr*, and maximum data length, written to *bufcnt*, of a telemetry packet, available for use by subclasses of this type of packet. The actual used length is set by setLength().

Concurrency: Guarded

19.6.3 setCmdId()

Public member of: **`TfCmdResponse`**

Return Class: **`void`**

Arguments:

`unsigned` *`cmdid`*

Documentation:

This function sets the command identifier, *`cmdid`*, in the telemetry packet buffer.

Concurrency: Guarded

19.6.4 setTimestamp()

Public member of: **`TfCmdResponse`**

Return Class: **`void`**

Arguments:

`unsigned` *`tickcnt`*

Documentation:

This function stores the BEP timer tick counter, *`tickcnt`*, into the telemetry packet buffer.

Concurrency: Guarded

19.6.5 setLength()

Protected member of: **`TfCmdResponse`**

Return Class: **`void`**

Arguments:

`unsigned` *`wordcnt`*

Documentation:

This function sets the number of data words, *`wordcnt`*, written by the subclass to the telemetry packet buffer. The maximum available is obtained using `getBufferInfo`().

Concurrency: Guarded

19.7 Class TfDeaHouse

Documentation:

This class formats DEA Housekeeping telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Implementation Uses:

`TlmAllocator` *`deaHouseAllocator`*

Public Interface:

Operations:
`addEntry()`
`getMaxEntryCount()`
`setIdInfo()`

Private Interface:

Has-A Relationships:

`unsigned` *`nxtentry`*:: This is the index of the next DEA Housekeeping entry slot available for use.

Concurrency: Guarded

Persistence: Transient

19.7.1 addEntry()

Public member of: **TfDeaHouse**

Return Class: **Boolean**

Arguments:

DeaHouseId *entryid*
unsigned *value*

Documentation:

This function appends a DEA Housekeeping Identifier/Value pair, *entryid* and *value* respectively, to the end of the telemetry packet. This function returns *BoolTrue* if the value was added, and *BoolFalse* if the telemetry packet is full. NOTE: This type of telemetry packet is not bit-packed.

Concurrency: Guarded

19.7.2 getMaxEntryCount()

Public member of: **TfDeaHouse**

Return Class: **unsigned**

Documentation:

This function returns the maximum number of DEA Housekeeping Identifier/Value pairs which can be contained within a DEA Housekeeping packet.

Concurrency: Guarded

19.7.3 setIdInfo()

Public member of: **`TfDeaHouse`**

Return Class: **`void`**

Arguments:

`unsigned` *`blockid`*
`unsigned` *`tickcount`*

Documentation:

This function sets the parameter block id, *`blockid`*, used to configure the DEA housekeeping run, and the BEP's tick counter, *`tickcount`*, at about the time the acquisition for this packet started.

Concurrency: Guarded

19.8 Class TfDump

Documentation:

This class is responsible for forming the common portion of parameter dump telemetry packet buffers.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Interface:

Operations: `TfDump()`
`setBlockId()`

Protected Interface:

Operations: `storeBlock()`

Concurrency: Guarded

Persistence: Transient

19.8.1 TfDump()

Public member of: **`TfDump`**

Arguments:

`TlmAllocator*` *`allocator`*
`TlmFormatTag` *`format`*

Documentation:

This constructor associates the parameter dump class with the telemetry buffer manager, *`allocator`*, and the telemetry format tag, *`format`*.

Concurrency: Guarded

19.8.2 setBlockId()

Public member of: **TfDump**

Return Class: **void**

Arguments:

unsigned *blockid*

Documentation:

This function stores the parameter block identifier, *blockid*, in the telemetry buffer header.

Concurrency: Guarded

19.8.3 storeBlock()

Protected member of: **TfDump**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*
unsigned *wordcnt*

Documentation:

This function provides a common function used by subclasses to copy a parameter block into the body of the telemetry packet buffer. *srcaddr* is the address of the parameter block, specified in terms of an array of 32-bit words. *wordcnt* is the number of 32-bit words to copy into the telemetry packet buffer.

Postconditions:

The number of 32-bit words must fit entirely within the available packet buffer space.

Concurrency: Guarded

19.9 Class TfDump1dWin

Documentation:

This class is responsible for forming the body of a 1-D Window List Parameter Block telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfDump`**

Implementation Uses:

`TlmAllocator` *`scienceAllocator`*

Public Interface:

Operations: `copyBlock()`

Concurrency: Guarded

Persistence: Transient

19.9.1 copyBlock()

Public member of: **`TfDump1dWin`**

Return Class: **`void`**

Arguments:

`const Pb1dWindow&` *`pblock`*

Documentation:

This function copies the parameter block, referenced by *`pblock`*, into the telemetry packet buffer. This function uses **`TfDump`**::`storeBlock`() to perform the actual copy.

Concurrency: Guarded

19.10 Class TfDump2dWin

Documentation:

This class is responsible for forming the body of a 2-D Window List Parameter Block telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfDump`**

Implementation Uses:

`TlmAllocator` *`scienceAllocator`*

Public Interface:

Operations: `copyBlock()`

Concurrency: Guarded

Persistence: Transient

19.10.1 copyBlock()

Public member of: **`TfDump2dWin`**

Return Class: **`void`**

Arguments:

`const Pb2dWindow&` *`pblock`*

Documentation:

This function copies the parameter block, referenced by *`pblock`*, into the telemetry packet buffer. This function uses **`TfDump`**::`storeBlock`() to perform the actual copy.

Concurrency: Guarded

19.11 Class TfDumpCc

Documentation:

This class is responsible for forming the body of a Continuous Clocking Parameter Block telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfDump`**

Implementation Uses:

`TlmAllocator` *`scienceAllocator`*

Public Interface:

Operations: `copyBlock()`

Concurrency: Guarded

Persistence: Transient

19.11.1 copyBlock()

Public member of: **`TfDumpCc`**

Return Class: **`void`**

Arguments:

`const PbContClock&` *`pblock`*

Documentation:

This function copies the parameter block, referenced by *`pblock`*, into the telemetry packet buffer. This function uses **`TfDump`**::`storeBlock`() to perform the actual copy.

Concurrency: Guarded

19.12 Class TfDumpDeaHouse

Documentation:

This class is responsible for forming the body of a DEA Housekeeping Parameter Block telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfDump`**

Implementation Uses:

`TlmAllocator` *`deaHouseAllocator`*

Public Interface:

Operations: `copyBlock()`

Concurrency: Guarded

Persistence: Transient

19.12.1 copyBlock()

Public member of: **`TfDumpDeaHouse`**

Return Class: **`void`**

Arguments:

`const PbDeaHouse&` *`pblock`*

Documentation:

This function copies the parameter block, referenced by *`pblock`*, into the telemetry packet buffer. This function uses **`TfDump`**::`storeBlock`() to perform the actual copy.

Concurrency: Guarded

19.13 Class TfDumpTe

Documentation:

This class is responsible for forming the body of a Timed Exposure Parameter Block telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfDump`**

Implementation Uses:

`TlmAllocator` *`scienceAllocator`*

Public Interface:

Operations: `copyBlock()`

Concurrency: Guarded

Persistence: Transient

19.13.1 copyBlock()

Public member of: **`TfDumpTe`**

Return Class: **`void`**

Arguments:

`const PbTimedExp&` *`pblock`*

Documentation:

This function copies the parameter block, referenced by *`pblock`*, into the telemetry packet buffer. This function uses **`TfDump`**::`storeBlock`() to perform the actual copy.

Concurrency: Guarded

19.14 Class TfExecBep

Documentation:

This class formats the telemetered response to an “Execute BEP Memory” command.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations: `setReturnValue()`

Concurrency: Guarded

Persistence: Transient

19.14.1 setReturnValue()

Public member of: **`TfExecBep`**

Return Class: **`void`**

Arguments:

`unsigned` *`value`*

Documentation:

This function stores the result of the function call, *`value`*, into the telemetry packet buffer.

Concurrency: Guarded

19.15 Class TfExecFep

Documentation:

This class formats the telemetry response to an “Execute Front End Memory” command.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations:
`setFepId()`
`setReturnValue()`

Concurrency: Guarded

Persistence: Transient

19.15.1 setFepId()

Public member of: **`TfExecFep`**

Return Class: **`void`**

Arguments:

`FepId` *`fepid`*

Documentation:

This function stores the Front End Processor identifier, specified by *`fepid`*, into the telemetry packet buffer.

Concurrency: Guarded

19.15.2 setReturnValue()

Public member of: **`TfExecFep`**

Return Class: **`void`**

Arguments:

`unsigned` *`value`*

Documentation:

This function stores the result of the function call, *`value`*, into the telemetry packet buffer.

Concurrency: Guarded

19.16 Class TfReadBep

Documentation:

This class formats a “Read Back End Memory” telemetry packet, and provides client code with the data buffer to store the read data (or code).

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations:
`getReadBuffer()`
`setMemAddr()`
`setReadLength()`

Concurrency: Guarded

Persistence: Transient

19.16.1 getReadBuffer()

Public member of: **`TfReadBep`**

Return Class: **`void`**

Arguments:

`unsigned*&` *`readbuf`*
`unsigned&` *`readcnt`*

Documentation:

This function stores the start of the read memory buffer in *`readbuf`*, and the maximum number of words that can be stored in *`readcnt`*.

Concurrency: Guarded

19.16.2 setMemAddr()

Public member of: **TfReadBep**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*

Documentation:

This function stores the starting memory location, *srcaddr*, of the data contained within the packet in the telemetry buffer.

Concurrency: Guarded

19.16.3 setReadLength()

Public member of: **TfReadBep**

Return Class: **void**

Arguments:

unsigned *readcnt*

Documentation:

This function sets the packet length based on the number of 32-bit words stored in the packet's read buffer, *readcnt*.

Concurrency: Guarded

19.17 Class TfReadFep

Documentation:

This class formats a “Read Front End Memory” telemetry packet, and provides client code with the data buffer to store the read data (or code).

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations:

```
getReadBuffer()
setFepId()
setMemAddr()
setReadLength()
```

Concurrency: Guarded

Persistence: Transient

19.17.1 getReadBuffer()

Public member of: **TfReadFep**

Return Class: **void**

Arguments:

unsigned*& *readbuf*
unsigned& *readcnt*

Documentation:

This function supplies the caller with the address and capacity of the packet's read data buffer. It sets *readbuf* to the starting address of the buffer to use, and sets *readcnt* to the maximum number of words that can be stored in that buffer.

Concurrency: Guarded

19.17.2 setFepId()

Public member of: **TfReadFep**

Return Class: **void**

Arguments:

FepId *fepid*

Documentation:

This function stores the Front End Processor identifier, specified by *fepid*, into the telemetry packet buffer.

Concurrency: Guarded

19.17.3 setMemAddr()

Public member of: **TfReadFep**

Return Class: **void**

Arguments:

const unsigned* *srcaddr*

Documentation:

This function stores the starting memory location, *srcaddr*, of the data contained within the packet in the telemetry buffer.

Concurrency: Guarded

19.17.4 setReadLength()

Public member of: **TfReadFep**

Return Class: **void**

Arguments:

unsigned *readcnt*

Documentation:

This function sets the packet length based on the number of 32-bit words stored in the packet's read buffer, *readcnt*.

Concurrency: Guarded

19.18 Class TfReadPram

Documentation:

This class formats “Read DEA PRAM” telemetry packets, and provides its client with the data buffer address and maximum length.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations:

```
getReadBuffer()
setBoardId()
setIndex()
setReadCount()
```

Concurrency: Guarded

Persistence: Transient

19.18.1 getReadBuffer()

Public member of: **TfReadPram**

Return Class: **void**

Arguments:

DeaPramWord*& *readbuf*
unsigned& *wordcnt*

Documentation:

This function returns the starting address and maximum entry count of the telemetry packet's DEA PRAM word buffer. It writes the starting address to *readbuf*, and it writes the maximum number of **DeaPramWord**s that can be stored into the buffer to *wordcnt*.

Concurrency: Guarded

19.18.2 setBoardId()

Public member of: **TfReadPram**

Return Class: **void**

Arguments:

DeaCcdBdId *board*

Documentation:

This function sets the DEA CCD Controller Board Identifier, *board*, whose PRAM contents are being stored in the telemetry packet.

Concurrency: Guarded

19.18.3 setIndex()

Public member of: **TfReadPram**

Return Class: **void**

Arguments:

unsigned *index*

Documentation:

This function sets the PRAM offset, *index*, of the first PRAM word being stored into this packet.

Concurrency: Guarded

19.18.4 setReadCount()

Public member of: **TfReadPram**

Return Class: **void**

Arguments:

unsigned *wordcnt*

Documentation:

This function stores the number of PRAM words, *count*, written into the packet, into the telemetry packet buffer.

Concurrency: Guarded

19.19 Class TfReadSram

Documentation:

This class formats “Read DEA SRAM” telemetry packets, and provides its client with the data buffer address and maximum length.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfCmdResponse`**

Implementation Uses:

`TlmAllocator` *`memServerAllocator`*

Public Interface:

Operations:

```
getReadBuffer()
setBoardId()
setIndex()
setReadCount()
```

Concurrency: Guarded

Persistence: Transient

19.19.1 getReadBuffer()

Public member of: **TfReadSram**

Return Class: **void**

Arguments:

DeaSramWord*& *readbuf*
unsigned *wordcnt*

Documentation:

This function returns the starting address and maximum entry count of the telemetry packet's DEA SRAM word buffer. It writes the starting address to *readbuf*, and it writes the maximum number of **DeaSramWord**s that can be stored into the buffer to *wordcnt*.

Concurrency: Guarded

19.19.2 setBoardId()

Public member of: **TfReadSram**

Return Class: **void**

Arguments:

DeaCcdBdId *board*

Documentation:

This function sets the DEA CCD Controller Board Identifier, *board*, whose SRAM contents are being stored in the telemetry packet.

Concurrency: Guarded

19.19.3 setIndex()

Public member of: **TfReadSram**

Return Class: **void**

Arguments:

unsigned *index*

Documentation:

This function sets the SRAM offset, *index*, of the first SRAM word being stored into this packet.

Concurrency: Guarded

19.19.4 setReadCount()

Public member of: **TfReadSram**

Return Class: **void**

Arguments:

unsigned *count*

Documentation:

This function stores the number of SRAM words, *count*, written into the packet, into the telemetry packet buffer.

Concurrency: Guarded

19.20 Class TfSciBias

Documentation:

This class is responsible for packing bias-map data into telemetry packet buffers.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Uses:

`HuffmanTable`

Public Interface:

Operations: `packBiasMap()`

Concurrency: Guarded

Persistence: Transient

19.20.1 packBiasMap()

Public member of: **`TfSciBias`**

Concurrency: Guarded

19.21 Class TfSciDaCcFaint

Documentation:

This class formats the body of a Continuous Clocking Faint Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.22 Class TfSciDaCcFaintBias

Documentation:

This class formats the body of a Continuous Clocking Faint-with-Bias Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.23 Class TfSciDaCcGraded

Documentation:

This class formats the body of a Continuous Clocking Graded Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.24 Class TfSciDaCcRaw

Documentation:

This class formats the body of a Continuous Clocking Raw Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.25 Class TfSciDaTeFaint

Documentation:

This class formats the body of a Timed Exposure Faint Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Public Interface:

Operations: `packEvents()`

Concurrency: Guarded

Persistence: Transient

19.25.1 packEvents()

Public member of: **`TfSciDaTeFaint`**

Concurrency: Guarded

19.26 Class TfSciDaTeFaintBias

Documentation:

This class formats the body of a Timed Exposure Faint-with-Bias Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.27 Class TfSciDaTeGraded

Documentation:

This class formats the body of a Timed Exposure Graded Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.28 Class TfSciDaTeHist

Documentation:

This class formats the body of a Timed Exposure Histogram Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Concurrency: Guarded

Persistence: Transient

19.29 Class TfSciDaTeRaw

Documentation:

This class formats the body of a Timed Exposure Raw Mode data telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciData`**

Public Interface:

Operations: `packData()`

Concurrency: Guarded

Persistence: Transient

19.29.1 packData()

Public member of: **`TfSciDaTeRaw`**

Concurrency: Guarded

19.30 Class TfSciData

Documentation:

This class is responsible for formatting the common header portion of all science data telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Interface:

Operations: `TfSciData()`
`setDataInfo()`

Concurrency: Guarded

Persistence: Transient

19.30.1 TfSciData()

Public member of: **`TfSciData`**

Arguments:

`TlmFormatTag` *`format`*

Documentation:

This constructor associates the built instance with the telemetry format, specified by *`format`*.

Concurrency: Guarded

19.30.2 setDataInfo()

Public member of: **`TfSciData`**

Return Class: **`void`**

Arguments:

`CcdId` *`ccdid`*
`unsigned` *`dataseq`*

Documentation:

This function stores the passed information into the common science data packet header area of the telemetry buffer. *`ccdid`* identifies which CCD produced the stored data, and *`dataseq`* is the sequence number of the science data packet for the current exposure.

Concurrency: Guarded

19.31 Class TfSciErCcEvent

Documentation:

This class formats exposure records for Continuous Clocking event-based modes telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErEvent`**

Concurrency: Guarded

Persistence: Transient

19.32 Class TfSciErCcFaint

Documentation:

This class formats the exposure record for a Continuous Clocking Faint Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErCcEvent`**

Concurrency: Guarded

Persistence: Transient

19.33 Class TfSciErCcFaintBias

Documentation:

This class formats the exposure record for a Continuous Clocking Faint-with-Bias Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErCcEvent`**

Concurrency: Guarded

Persistence: Transient

19.34 Class TfSciErCcGraded

Documentation:

This class formats the exposure record for a Continuous Clocking Graded Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErCcEvent`**

Concurrency: Guarded

Persistence: Transient

19.35 Class TfSciErCcRaw

Documentation:

This class formats the exposure record for a Continuous Clocking Raw Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciExpRecord`**

Concurrency: Guarded

Persistence: Transient

19.36 Class TfSciErEvent

Documentation:

This class formats the exposure record for a Timed Exposure Faint Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciExpRecord`**

Public Interface:

Operations:
```
setAboveThresholdCnt()
setBiasParityErrorCnt()
setDiscardGradeCnt()
setDiscardPhCnt()
setDiscardWinCnt()
setEventCnt()
setOverclockLevels()
```

Concurrency: Guarded

Persistence: Transient

19.36.1 setAboveThresholdCnt()

Public member of: **`TfSciErEvent`**

Documentation:

This function stores the number of pixels detected above threshold during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.36.2 setBiasParityErrorCnt()

Public member of: **TfSciErEvent**

Documentation:

This function stores the number of parity errors detected since the start of the science run into the telemetry packet buffer.

Concurrency: Guarded

19.36.3 setEventCnt()

Public member of: **TfSciErEvent**

Documentation:

This function stores the events telemetered during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.36.4 setDiscardGradeCnt()

Public member of: **TfSciErEvent**

Documentation:

This function stores the number of events discarded due to their grade code during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.36.5 setDiscardPhCnt()

Public member of: **TfSciErEvent**

Documentation:

This function stores the number of events discarded due to their pulse height during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.36.6 setDiscardWinCnt()

Public member of: **`TfSciErEvent`**

Documentation:

This function stores the number of events discarded by the window list filter during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.36.7 setOverclockLevels()

Public member of: **`TfSciErEvent`**

Documentation:

This function stores the delta-overclock values used during the exposure into the telemetry packet buffer.

Concurrency: Guarded

19.37 Class TfSciErTeEvent

Documentation:

This class formats exposure records for Timed Exposure event-based modes telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErEvent`**

Concurrency: Guarded

Persistence: Transient

19.38 Class TfSciErTeFaint

Documentation:

This class formats the exposure record for a Timed Exposure Faint Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErTeEvent`**

Concurrency: Guarded

Persistence: Transient

19.39 Class TfSciErTeFaintBias

Documentation:

This class formats the exposure record for a Timed Exposure Faint-with-Bias Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErTeEvent`**

Public Interface:

Operations: `setBiasOffset()`

Concurrency: Guarded

Persistence: Transient

19.39.1 setBiasOffset()

Public member of: **`TfSciErTeFaintBias`**

Concurrency: Guarded

19.40 Class TfSciErTeGraded

Documentation:

This class formats the exposure record for a Timed Exposure Graded Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciErTeEvent`**

Concurrency: Guarded

Persistence: Transient

19.41 Class TfSciErTeHist

Documentation:

This class formats the exposure record for a Timed Exposure Histogram Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciExpRecord`**

Concurrency: Guarded

Persistence: Transient

19.42 Class TfSciErTeRaw

Documentation:

This class formats the exposure record for a Timed Exposure Raw Mode telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TfSciExpRecord`**

Public Interface:

Operations: `setPixelCount()`

Concurrency: Guarded

Persistence: Transient

19.42.1 setPixelCount()

Public member of: **`TfSciErTeRaw`**

Return Class: **`void`**

Arguments:

`unsigned` *`pixelcnt`*

Documentation:

This function stores the number of pixels sent by this exposure's data packet sequence.

Concurrency: Guarded

19.43 Class TfSciExpRecord

Documentation:

This class formats the common header section of all Science Exposure Record telemetry packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Interface:

Operations: `TfSciExpRecord()`
`setExposureInfo()`

Concurrency: Guarded

Persistence: Transient

19.43.1 TfSciExpRecord()

Public member of: **`TfSciExpRecord`**

Arguments:

`TlmFormatTag` *`format`*

Documentation:

This constructor associates the format tag, *`format`*, with the exposure record instance.

Concurrency: Guarded

19.43.2 setExposureInfo()

Public member of: **TfSciExpRecord**

Return Class: **void**

Arguments:

unsigned *pblock*
unsigned *starttime*
CcdId *ccdid*
unsigned *expnum*

Documentation:

This function stores the passed information into the common exposure header portion of the telemetry packet buffer. *ccdid* refers to the CCD producing the exposure. *expnum* is the exposure number. *pblock* is the parameter block identifier used to configure the run, and *starttime* is the DEA-latched timestamp at the moment the CCD sequencers were started at the beginning of the run.

Concurrency: Guarded

19.44 Class TfSciReport

Documentation:

This class is responsible for forming the post-science run report telemetry packet.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Concurrency: Guarded

Persistence: Transient

19.45 Class TfStartup

Documentation:

This class formats Back End Processor system startup message packets.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Implementation Uses:

`TlmAllocator` *`startupAllocator`*

Public Interface:

Operations:

```
setBlocksValidFlag()
setBootReason()
setPatchValidFlag()
setSysConfigValidFlag()
setTimestamp()
setVersion()
```

Concurrency: Guarded

Persistence: Transient

19.45.1 setBlocksValidFlag()

Public member of: **TfStartup**

Return Class: **void**

Arguments:

Boolean *blockValid*

Documentation:

This function sets the “parameter blocks valid” flag in the telemetry packet buffer indicating whether or not the Parameter Block CRCs are intact. *blockValid* is *BoolTrue* if the none of the parameter blocks have been corrupted, and *BoolFalse* if one or more of the blocks have been corrupted.

Concurrency: Guarded

19.45.2 setBootReason()

Public member of: **TfStartup**

Return Class: **void**

Arguments:

BootMode *bootMode*
FatalCode *lastFatalCode*
unsigned *lastFatalValue*

Documentation:

This function sets the reason for the reboot, and includes the last sent fatal error message code and argument. *bootMode* indicates the reset mode, and *lastFatalCode* and *lastFatalValue* indicate the last fatal error message sent by the instrument since power-up.

Concurrency: Guarded

19.45.3 setPatchValidFlag()

Public member of: **TfStartup**

Return Class: **void**

Arguments:

Boolean *patchValid*

Documentation:

This function sets the “patch list valid” flag in the telemetry packet buffer, indicating whether or not the PatchList CRC is intact. *patchValid* is *BoolTrue* if the list is not corrupted, and *BoolFalse* if the list was corrupted.

Concurrency: Guarded

19.45.4 setSysConfigValidFlag()

Public member of: **TfStartup**

Return Class: **void**

Arguments:

Boolean *sysConfigValid*

Documentation:

This function sets the “system configuration valid” flag in the telemetry packet buffer, indicating whether or not the System Configuration Parameter Block CRC is intact. *sysConfigValid* is *BoolTrue* if the block is not corrupted, and *BoolFalse* if the block was corrupted.

Concurrency: Guarded

19.45.5 setTimestamp()

Public member of: **`TfStartup`**

Return Class: **`void`**

Arguments:

`unsigned` *`tickcount`*

Documentation:

This function stores the passed BEP tick counter, *`tickcount`*, into the packet buffer.

Concurrency: Guarded

19.45.6 setVersion()

Public member of: **`TfStartup`**

Return Class: **`void`**

Arguments:

`unsigned` *`version`*

Documentation:

This function writes the software version code, *`version`*, into the telemetry packet and sets the telemetry packet buffer length.

Concurrency: Guarded

19.46 Class TfSwHouse

Documentation:

This class represents a software housekeeping telemetry packet. It is responsible for formatting and accumulating housekeeping information within the packet and for posting the packet to the telemetry manager.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Implementation Uses:

`TlmAllocator` *`swHouseAllocator`*
`IntrGuard`

Public Interface:

Operations:
`accumulateStat()`
`setupBuffer()`

Private Interface:

Has-A Relationships:

`const unsigned` *`datalen`*: This value, set during construction, specifies the total data space needed by the housekeeper for its statistics and header.

Concurrency: Guarded

Persistence: Transient

19.46.1 accumulateStat()

Public member of: **TfSwHouse**

Return Class: **void**

Arguments:

SwStatistic *statid*
unsigned *value*

Documentation:

If the form has a packet, this function increments the statistic value associated with *statid* and overwrites the statistic's information field with *value*. If the form does not have a packet, the statistic report is ignored.

Concurrency: Synchronous

19.46.2 setupBuffer()

Public member of: **TfSwHouse**

Return Class: **Boolean**

Arguments:

unsigned *tickcnt*

Documentation:

This function attempts to acquire a telemetry packet buffer from its *allocator*, and if successful, prepares the body of the telemetry buffer to record new software housekeeping information. If a buffer is successfully obtained, it sets the length of the packet, stores *tickcnt* into the housekeeping header, and returns *BoolTrue*. Otherwise, it returns *BoolFalse*.

Concurrency: Guarded

20.0 Parameter Block Management (36-53229 01)

20.1 Purpose

The purpose of the Parameter Block Management classes are to maintain sets of science and DEA housekeeping parameter blocks.

This section describes two main classes, the **`PblockList`** class, the **`Pblock`** class, and lists their respective subclasses. Detailed descriptions of each of the subclasses are provided in Appendix TBD.

20.2 Uses

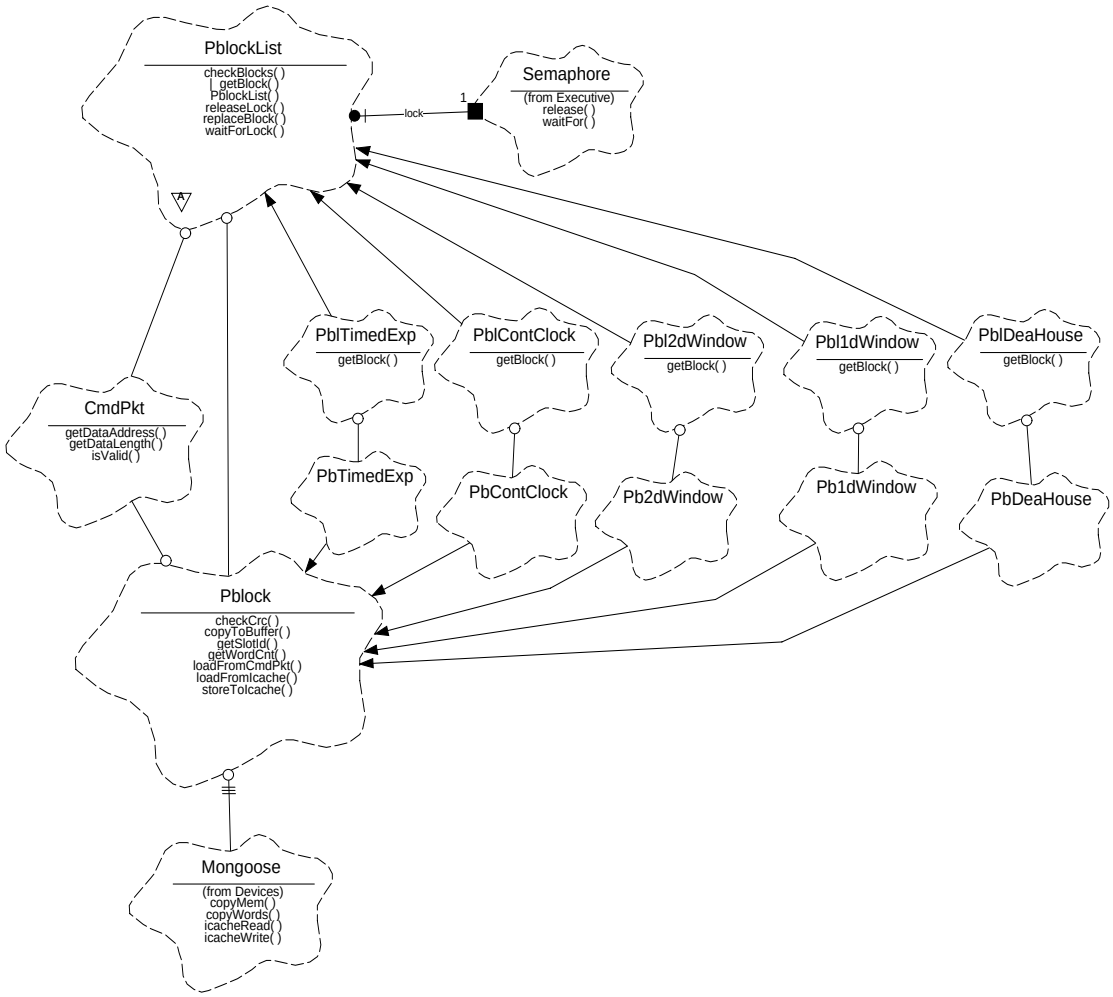
The following lists the primary uses of the Parameter Block Management classes:

Use 1:: Store parameter blocks for use by science and DEA housekeeping managers
Use 2:: Perform integrity checks on parameter blocks
Use 3:: Supply parameter blocks when needed for a science or DEA housekeeping run

20.3 Organization

Figure 89 illustrates the class relationships used by the Parameter Block Management classes.

FIGURE 89. Parameter Block Management Classes



`Pblock`- This abstract class is responsible for interpreting the contents of a single parameter block, and for moving parameter data into and out of storage in Instruction Cache RAM. It provides functions to load the contents of its internal buffer from the contents of a command packet (`loadFromCmdPkt`), to copy its contents into a client supplied buffer (`copyToBuffer`), to store and retrieve the contents of its internal buffer to and from a region of instruction cache memory (`storeToIcache` and `loadFromIcache`), to verify the integrity of the block using its CRC value (`checkCrc`), and to retrieve the slot id and number of words contained within the block (`getSlotId` and `getWordCnt`). This class uses the services provided by the ***`Devices`***::**`Mongoose`** device class to copy data, and to store and load data from instruction cache RAM.

`PblockList`- This abstract class is responsible for maintaining a fixed size collection of homogenous parameter blocks (**`Pblock`** or one of its subclasses). This class provides functions to replace the contents of one its parameter blocks with the contents of a command packet (`replaceBlock`), to check the integrity of all of its blocks (`checkBlocks`) and to retrieve the contents of one of its blocks (`getBlock`). It also provides functions to obtain and release a semaphore associated with the collection

(`waitForLock` and `releaseLock`). This class's constructor (`PblockList` not shown) is responsible for initializing the list's instance variables and checking their validity. This class uses the services provided by the **`Pblock`** class to manage the contents of a single parameter block, and contains a ***`Executive`***::**`Semaphore`** instance associated with its collection of parameter blocks.

The following lists the various subclasses of **`Pblock`**. Each of these classes are responsible for providing functions with return the values of the various fields within the parameter block. The functions belonging to each of these classes are produced using a code-generator directly from the Instrument Program and Command List specification. The functions belonging to each of these classes are described in Appendix TBD:

`PbTimedExp` - Timed Exposure Parameter Block

`PbContClock` - Continuous Clocking Parameter Block

`Pb2dWindow` - 2D Window List Parameter Block

`Pb1dWindow` - 1D Window List Parameter Block

`PbDeaHouse` - DEA Housekeeping Parameter Block

The following lists the various subclasses of **`PblockList`**. There is one global instance of each of these classes within the instrument. Each of these classes is responsible for managing the collection of parameter blocks of the indicated type, and, in addition to the member functions provided by **`PblockList`**, overload the `getBlock`() function with a public member function. Rather than using the generic **`Pblock`** type as an argument, the overloaded functions require the correct parameter block type for their input arguments.

`PblTimedExp` -Manage all Timed Exposure Parameter Blocks (**`PbTimedExp`**)

`PblContClock` - Manage all Continuous Clocking Parameter Blocks (**`PbContClock`**)

`Pbl2dWindow` -Manage all 2D Window List Parameter Blocks (**`Pb2dWindow`**)

`Pbl1dWindow`- Manage all 1D Window List Parameter Blocks (**`Pb1dWindow`**)

`PblDeaHouse` - Manage all DEA Housekeeping Parameter Blocks (**`PbDeaHouse`**)

20.4 Parameter Block Storage

The **`Pblock`** class and its various subclasses are used as temporary objects which move data into and out of I-cache slots, and provide functions which retrieve specific fields of a given type of parameter block. Each parameter block slot in I-cache is sized to hold the maximum size allowed for a parameter block (i.e. one command packet). The number of slots reserved for each type of block is TBD. Upon power-on, this area of I-cache is loaded with default parameter blocks from the main instrument ROM. Subsequent resets do not modify the slot contents. Assuming, for now, that the instrument supports 4 blocks of each type, Table 22 illustrates the overall layout of parameter block slots within I-cache RAM, where the addresses and sizes are gross approximations. The first two slots of each type of block contain the default parameter blocks for Imaging and Spectroscopy observations.

TABLE 22. I-cache Parameter Block Layout (TBD)

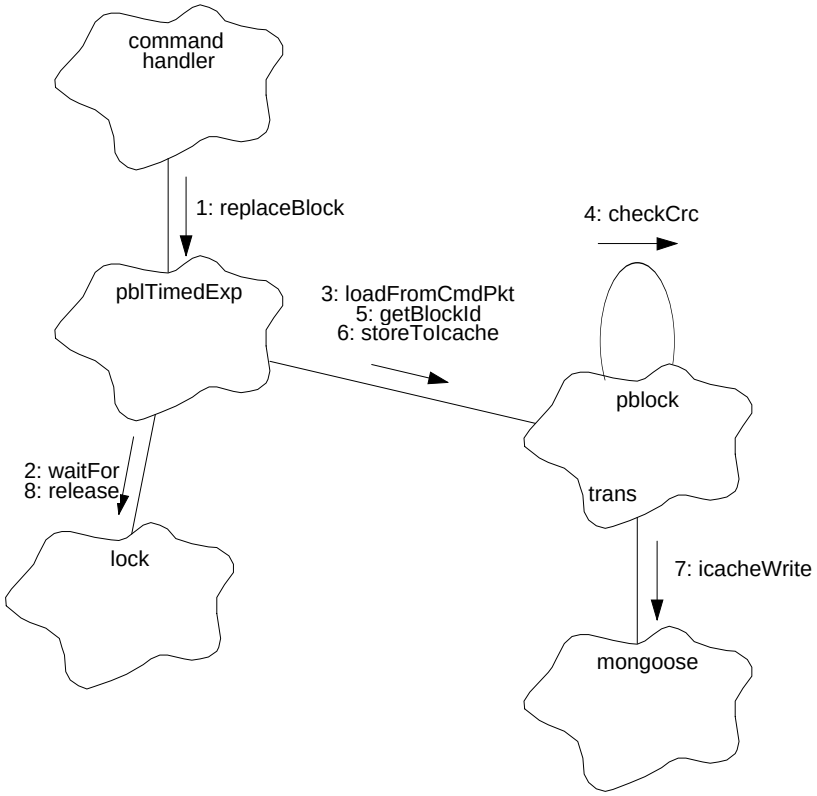
Region	Address	Byte Size	Description	Slot #
Patch Area (grow down from 0x800fffff)	0x800fffff 0x800cfc00	0x30400		
Bad Pixel and Column Maps	0x800c2c00	0xd000		
DEA Housekeeping Blocks	0x800c2a00	0x200	General Purpose	3
	0x800c2800	0x200	General Purpose	2
	0x800c2600	0x200	General Purpose	1
	0x800c2400	0x200	Default	0
1D Window Blocks	0x800c2200	0x200	General Purpose	3
	0x800c2000	0x200	General Purpose	2
	0x800c1e00	0x200	Default Spectroscopy	1
	0x800c1c00	0x200	Default Imaging	0
2D Window Blocks	0x800c1a00	0x200	General Purpose	3
	0x800c1800	0x200	General Purpose	2
	0x800c1600	0x200	Default Spectroscopy	1
	0x800c1400	0x200	Default Imaging	0
Continuous Clocking Blocks	0x800c1200	0x200	General Purpose	3
	0x800c1000	0x200	General Purpose	2
	0x800c0e00	0x200	Default Spectroscopy	1
	0x800c0c00	0x200	Default Imaging	0
Timed Exposure Blocks	0x800c0a00	0x200	General Purpose	3
	0x800c0800	0x200	General Purpose	2
	0x800c0600	0x200	Default Spectroscopy	1
	0x800c0400	0x200	Default Imaging	0
System Configuration and Huffman Compression Tables	0x800c0000	0x400		
Code	0x80080000	0x40000		

20.5 Scenarios

20.5.1 Use 1: Store parameter blocks

Figure 90 illustrates the steps used to store Timed Exposure Parameter blocks. The steps used to store Continuous Clocking, 2D Window, 1D Window and DEA Housekeeping parameter blocks are identical except for the identities of the parameter block list objects.

FIGURE 90. Load Parameter Block Scenario



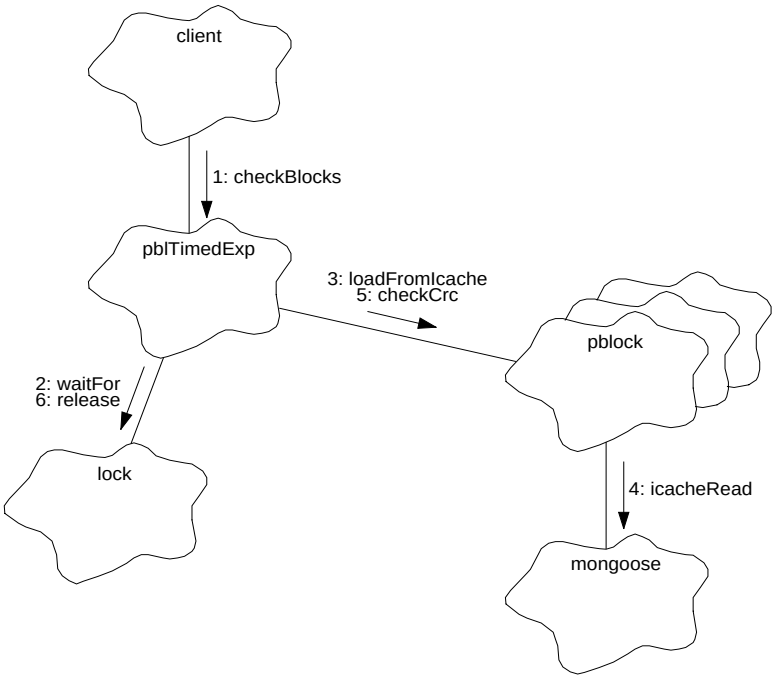
1. The client *commandHandler* receives a command to overwrite a parameter block, and instructs the Timed Exposure Parameter Block List object to store the block using *pblTimedExp*.replaceBlock().
2. replaceBlock() suspends the current task until it has obtained its semaphore, indicating that it has exclusive access to the parameter block list using *lock*.waitFor(). If the wait time-out expires, the attempt to replace the block is aborted and the function returns *BoolFalse*. For the purposes of this scenario, assume that the *lock* is obtained successfully and returns *BoolTrue*.
3. replaceBlock() declares an instance of a Timed Exposure Parameter Block, *pblock*, and instructs it to load its contents from the command packet, passed in from the handler using *pblock*.loadFromCmdPkt(). At this point, *pblock* has a copy of the parameter block data.

4. *pblock*.loadFromCmdPkt() copies the parameter data from the command buffer, and then checks the integrity of the copied data by calling checkCrc(). If the data has been corrupted, loadFromCmdPkt() returns an error. Assume for the purposes of this scenario that the parameters are intact.
5. Once *pblock* has copied the parameters, *pblTimedExp*.replaceBlock() obtains the slot identifier from the block using *pblock*.getSlotId().
6. replaceBlock() uses the returned slot id to select the region in Instruction Cache RAM to store the block, and copies the block data into I-cache using *pblock*.storeToIcache().
7. *pblock*.storeToIcache() uses *mongoose*.icacheWrite() to copy its parameter block data to the selected region.
8. Once the parameter block has been stored into the appropriate slot in I-cache, replaceBlock() releases its semaphore, using *lock*.release().

20.5.2 Use 2: Perform integrity checks

Figure 91 illustrates the steps used to check the integrity of the collection of Timed Exposure Parameter blocks. The steps used to check other types of blocks are identical, except for the identities of the parameter block list objects.

FIGURE 91. Check integrity of stored Parameter Blocks



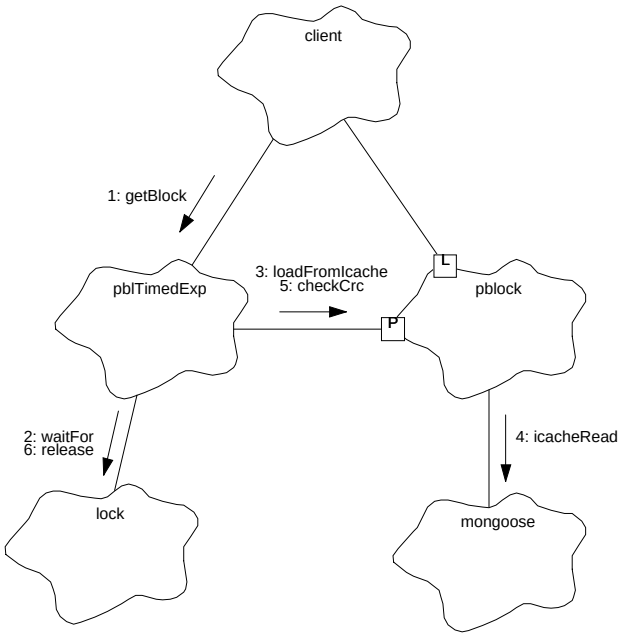
1. The *client* instructs the Timed Exposure Parameter Block List object to verify the integrity of each of its blocks using *pblTimedExp*.checkBlocks().
2. checkBlocks() prevents other tasks from overwriting the contents of the blocks by obtaining its semaphore using *lock*.waitFor(). If the wait time-out expires, the attempt to check the block is aborted and the function returns *BoolFalse*. Assume that the *lock* is obtained successfully.
3. checkBlocks() iterates through each parameter slot that it maintains. On each iteration, it declares a Timed Exposure Parameter block instance, *pblock*, and instructs it to load its contents from the parameter slot maintained in I-cache RAM, using *pblock*.loadFromIcache().
4. *pblock*.loadFromIcache() uses *mongoose*.icacheRead() to copy the parameter block data from I-cache RAM into its private buffer.
5. *pblTimedExp*.checkBlocks() instructs the block to compute its CRC, compare the value with the one contained within the parameter block data, and return the result of the comparison, using *pblock*.checkCrc().

6. *pblTimedExp*.checkBlocks() repeats these steps for each of its parameter block slots. Once all of the blocks have been checked, checkBlocks() releases the lock, using *lock*.release(). It then returns whether or not any of its parameter blocks have been corrupted (*BoolTrue* if the blocks are intact, and *BoolFalse* if the lock attempt failed or any block has been corrupted).

20.5.3 Use 3: Supply parameter blocks when needed

Figure 92 illustrates the steps used to retrieve the contents of a Timed Exposure Parameter block. The steps used to obtain Continuous Clocking, 2D Window, 1D Window and DEA Housekeeping parameter blocks are identical except for the identities of the parameter block list objects.

FIGURE 92. Retrieve contents of Parameter Block



1. During system startup, the instrument constructs the parameter block list, *pblTimedExp*, specifying the address and size of the block (not shown). Once running, the *client* declares an instance of a Timed Exposure Parameter Block object, *pblock*, and passes the block slot id and the instance to the Timed Exposure Parameter Block list to be loaded to *pblTimedExp*.getBlock().
2. *pblTimedExp*.getBlock() prevents overwrites of the list while performing the retrieval using *lock*.waitFor(). If the wait's time-out expires, the fetch is aborted, and returns *BoolFalse*. Assume for the purposes of this scenario that the lock is obtained, and the function returns *BoolTrue*.
3. getBlock() instructs the block to load its contents from I-cache RAM, using *pblock*.loadFromIcache().
4. loadFromIcache() uses *mongoose*.icacheRead() to copy the parameter data into its local buffer.
5. getBlock() instructs the block to check its CRC, using *pblock*.checkCrc().
6. getBlock() releases the semaphore, using *lock*.release(). It then returns to the client, informing it whether or not the loaded block's CRC was valid or not.

20.6 Class PblockList

Documentation:

This class represents a collection of parameter blocks. It defines the common functions defined for all types of parameter blocks.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

Pblock

Public Interface:

Operations:
```
PblockList()
checkBlocks()
releaseLock()
replaceBlock()
waitForLock()
```

Protected Interface:

Operations: `getBlock()`

Private Interface:

Has-A Relationships:

Semaphore *lock*: This is used lock access to the parameter blocks.

unsigned* const *slotBase*: This points to the starting location within I-cache RAM of the array of parameter block slots.

const unsigned *slotCnt*: This is the number of parameter blocks.

const unsigned *slotWordCnt*: This specifies the number of 32-bit words in each parameter block slot maintained by this instance.

Concurrency: Guarded

Persistence: Persistent

20.6.1 PblockList()

Public member of: **PblockList**

Arguments:

unsigned *semid*
unsigned* *base*
unsigned *nSlots*
unsigned *wordCnt*

Documentation:

This is the constructor for a parameter block list. *semid* is the RTX id for the semaphore the instance uses to synchronize access to the parameters, *base* is the starting address within I-cache used to maintain the list of blocks, *nSlots* is the number of parameter block slots maintained by the instance, and *wordCnt* is the number of 32-bit words in each parameter block slot.

Semantics:

Construct *lock* using *semid*. Copy *base to* slotBase, *nSlots* to *slotCnt* and *wordCnt* to *slotWordCnt*.

Concurrency: Sequential

20.6.2 checkBlocks()

Public member of: **PblockList**

Return Class: **Boolean**

Documentation:

This function checks the CRC of all parameter blocks in the collection. It returns *BoolTrue* if all are ok, and *BoolFalse* if there is an error in one or more of the blocks.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. For each slot, declare *pblock*, instruct it to load from i-cache address *slotBase* + (slot counter * *slotWordCnt*), using *pblock*.loadFromIcache(), then call *pblock*.checkCrc() and set a flag it if fails. Once all slots have been checked, release the lock using releaseLock(). If any CRC fails, return *BoolFalse*. If all of the CRC checks pass, return *BoolTrue*.

Concurrency: Synchronous

20.6.3 getBlock()

Protected member of: **PblockList**

Return Class: **Boolean**

Arguments:

unsigned *slotid*
Pblock& *pblock*

Documentation:

This function fills *pblock* with the parameter block data stored in the location identified by *slotid*. If successful, this function returns *BoolTrue*. If *slotid* is invalid, the wait for the lock times-out, or if the CRC of the requested block is invalid, this function returns *BoolFalse*.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. Instruct *pblock* to load its contents from I-cache address *slotBase* + (*slotid* * *slotWordCnt*) using *pblock*.loadFromIcache(), then call *pblock*.checkCrc(). Release the lock using releaseLock(). If the CRC failed, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Synchronous

20.6.4 releaseLock()

Public member of: **PblockList**

Return Class: **void**

Documentation:

This function releases the parameter block list, by calling *lock*.release(), allowing other tasks to use this instance.

Concurrency: Synchronous

20.6.5 replaceBlock()

Public member of: **PblockList**

Return Class: **Boolean**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function uses the *cmdpkt* to replace the contents of the block specified by the slot id contained within the command packet pointed to by *cmdpkt*. This function returns *BoolTrue* if the replacement succeeds, and *BoolFalse* if it fails.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. Declare a parameter block, *pblock*, and instruct it to load its contents from the command packet, using *pblock*.loadFromCmdPkt(). Check its CRC using *pblock*.checkCrc(). If the CRC is valid, retrieve the parameter block's slot id using *pblock*.getSlotId(), and store its contents into I-cache using *pblock*.storeToIcache(). Release the semaphore using releaseLock(). If the loaded parameter block's CRC check failed return *BoolFalse* without storing the block. If the block is intact and stored, the function returns *BoolTrue*.

Concurrency: Synchronous

20.6.6 waitForLock()

Public member of: **`PblockList`**

Return Class: **`Boolean`**

Arguments:

`unsigned` *`timeout`*

Documentation:

This function waits for the parameter block list to become available, and locks it by calling *`lock`*`.waitFor`(). If successful, this function returns *`BoolTrue`*. If *`timeout`* is reached first, it returns *`BoolFalse`*.

Concurrency: Synchronous

20.7 Class Pblock

Documentation:

This class is used to define a common top-level interface for all parameter blocks loaded into the instrument. This class provides mechanisms to load parameter block data from command packets, to store and retrieve parameter data from I-cache. Subclasses of this class provide functions which retrieve block-specific fields.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

CmdPkt

Implementation Uses:

Mongoose

Public Interface:

Operations:

```
checkCrc()
copyToBuffer()
getSlotId()
getWordCnt()
loadFromCmdPkt()
loadFromIcache()
storeToIcache()
```

Protected Interface:

Has-A Relationships:

unsigned *databuf*[128]: This is the data buffer which contains the parameter block.

Concurrency: Guarded

Persistence: Transient

20.7.1 checkCrc()

Public member of: **Pblock**

Return Class: **Boolean**

Documentation:

This function computes the CRC of the current block and compares it with the CRC contained within the block. If the CRC is invalid, the function returns *BoolFalse*, else it returns *BoolTrue*.

Preconditions:

The block contents must have been loaded either via loadFromCmdPkt() or loadFromIcache().

Concurrency: Guarded

20.7.2 copyToBuffer()

Public member of: **Pblock**

Return Class: **void**

Arguments:

unsigned* *dstptr*

Documentation:

This function copies the body of the parameter block to the buffer pointed to by *dstptr*. The caller must ensure that the destination buffer is large enough to hold the contents of the parameter block.

Preconditions:

The block contents must have been loaded either via loadFromCmdPkt() or loadFromIcache(). *dstptr* must not point into I-cache.

Concurrency: Guarded

20.7.3 getSlotId()

Public member of: **Pblock**

Return Class: **unsigned**

Documentation:

This function returns the slot id of the parameter block.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt`() or `loadFromIcache`().

Concurrency: Guarded

20.7.4 getWordCnt()

Public member of: **Pblock**

Return Class: **unsigned**

Documentation:

This function returns the length of the parameter block, in 32-bit words.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt`() or `loadFromIcache`().

Concurrency: Guarded

20.7.5 loadFromCmdPkt()

Public member of: **Pblock**

Return Class: **void**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function loads the parameter block contents from the command packet pointed to by *cmdpkt*.

Semantics:

Get the packet's data buffer address and length using *cmdpkt*->getDataAddress() and *cmdpkt*->getDataLength() respectively. Copy the data from the packet into the private data buffer, *databuf*, using *mongoose*.copyMem().

Concurrency: Guarded

20.7.6 loadFromIcache()

Public member of: **Pblock**

Return Class: **void**

Arguments:

const unsigned* *srcptr*

Documentation:

This function loads the parameter block from data store in I-cache at the address, *srcptr* into its data buffer, using *mongoose*.icacheRead().

Concurrency: Guarded

20.7.7 storeToIcache()

Public member of: **`Pblock`**

Return Class: **`void`**

Arguments:

`unsigned*` *`dstptr`*

Documentation:

This function stores the parameter block information at the I-cache address *`dstptr`*, using *`mongoose`*.`icacheWrite`().

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt`() or `loadFromIcache`().

Concurrency: Guarded

21.0 IPCL Code-Generator (36-53232.01 01)

21.1 Purpose

The purpose of the IPCL Code-Generator is to interpret the command and telemetry structure definitions provided in the “ACIS Instrument Program and Command List” (MIT-36-01410) database and produce C++ source code to read fields from command and parameter block buffers, and to format data into telemetry packet buffers.

21.2 Uses

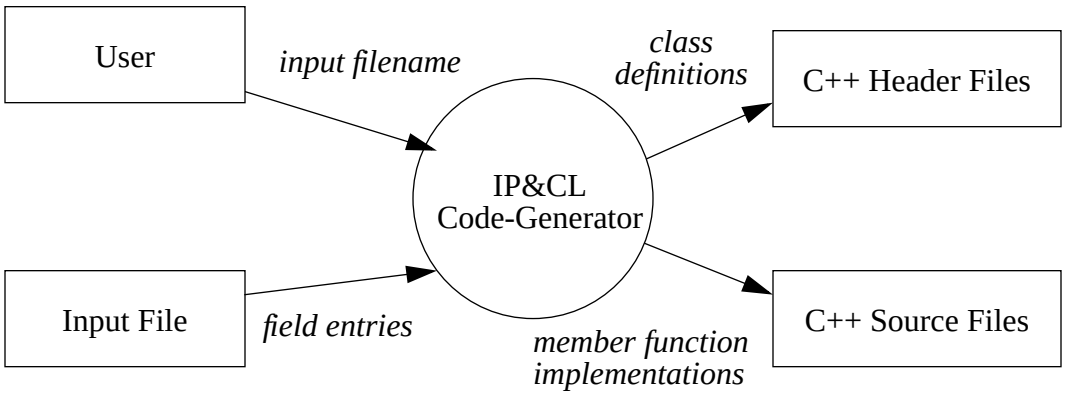
Use 1:: Write C++ classes which read fields from command packet buffers
Use 2:: Write C++ classes which write fields into telemetry packet buffers

21.3 Organization

The IPCL Code-Generator is written entirely in **Perl** and is contained in within three files: `ipcl_gen.pl`, `ipcl_reader.pl`, and `ipcl_writer.pl`. The code-generator reads in the IP&CL structures definition from a file containing lists of tab-separated values, with newlines separating entries in the database. The format of the IP&CL structures is described in Section 21.4 . The output of the code-generator is a set of C++ header and source files, one for each structure being read or written. Templates for the generated C++ files are described in Section 22.0 and Section 23.0 .

Figure 93 illustrates the main inputs and outputs of the IP&CL code-generator.

FIGURE 93. IPCL Generator Context Diagram

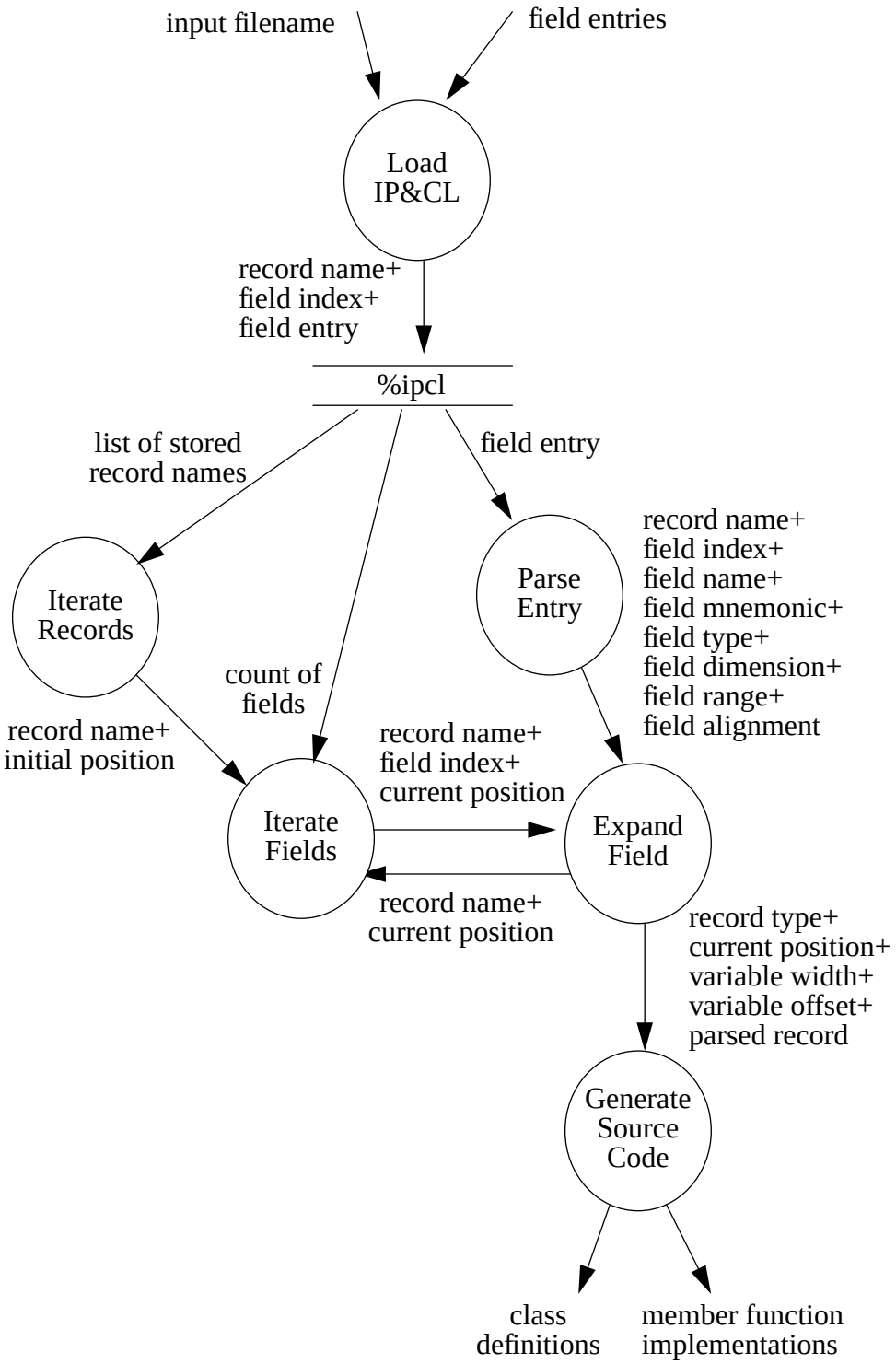


The user invokes the **Perl** IP&CL Code-Generator script passing it the name of the IP&CL Structures table file, *input filename*. The code generator opens and reads all of the *field entries* from the Input File. It then parses the *field entries*, and produces a series of C++ header and source files, one for each record structure defined in the input file. Each record structure is represented by a C++ class. The header files will contain the class definitions

for the defined classes, and the source files contain the member function implementations for these classes.

Figure 94 illustrates the Level-1 data flow of the IP&CL code-generator.

FIGURE 94. IPCL Generator Level-1 Data Flow Diagram



The following describes each data element illustrated in Figure 93 and Figure 94:

input filename - This identifies the file which contains the IP&CL Structure records to process.

field entries - These are the field definitions read from the IP&CL input file. Each entry contains a list of tab-separated ASCII elements, and ends with a newline character. See Section Section 21.4.2 for a description of the format of a field entry.

class definitions - These are the definitions for each generated C++ class.

member function implementations - These are the implementations for each member function of a class.

record name - This is the ASCII name of a record as defined in the IP&CL database. All fields belong to one record or another, and each *field entry* contains the name of the record to which it belongs. The code-generator uses the *record name* as part of a database key when indexing field entries. The code-generator includes a modified form of the *record name* when generating class names, and C++ header and source code filenames.

field index - Each field within a record is numbered. The code-generator uses the *field index* in its database key when indexing field entries.

%ipcl - This is a **Perl** associative array of IP&CL *field entry counts* and *field entries*. Array elements keyed using a *record name* alone return the number of fields defined for the named record. Elements which are keyed using a *record name* and *field index*, separated by the value of the **Perl** variable, *$;*, contain the *field entry* string, as read from the IP&CL input file.

```
$ipcl{record name} = count of fields for the named record
$ipcl{record name, field index} = field entry string
```

list of stored record names - This list is generated by asking **Perl** to produce an array of all database keys, and then sorts the list and prunes out any key containing the value of the *$;* **Perl** variable.

initial position - This item represents the initial bit-position of the current record being produced. This item's value is always 0.

count of fields - This represents the number of fields defined for a given record. It's value for a given record is contained in *$ipcl*{*record name*}.

current position - This item represents the current bit-position within a given record definition. It is used to compute the bit-position of a field or record, and to determine the need for padding if a field has a bit alignment requirement.

field name - This is the name of a field defined in the IP&CL database. The code-generator includes a modified form of the *field name* when generating member function names.

NOTE KLUDGE: On occasion, the generator also uses this name to identify fields which are intended to be grouped into an array. A set of two or more adjacent fields, which have the same base name, and whose names end with incrementing numbers starting from 0, may be treated as elements of an array by the code-generator.

field mnemonic - This is an 8-character mnemonic for the field in the IP&CL.NOTE KLUDGE: The code-generator uses the *mnemonic* of the first field of a record to determine whether the record defines a command packet structure, a telemetry packet structure or is an internal structure. If the last two characters of the mnemonic are “CI,” then the code generator treats the record as a command packet. If the last two characters are “XF,” then it treats the record as a telemetry packet. All others are treated as internal structures, and are ignored by the code-generator.

field type - This identifies the data type of the field. For ACIS software structures, a field has one of the following types:

int8 - signed, 8-bit integer

uint8 - unsigned, 8-bit integer

int16 - signed, 16-bit integer

uint16 - unsigned, 16-bit integer

int32 - signed, 32-bit integer

uint32 - unsigned, 32-bit integer

bit - bit-field where *field dimension* determines the number of bits in the field.

record name - The name of another record definition. The data type of the field is defined by the referenced record.

field dimension - This specifies the number of elements of *field type* in the field. If the *field dimension* is not 1, then the field represents an array of elements. If the *field dimension* is a constant expression, then the array is of fixed length. If the value of the *dimension* relies the value of another field within the record, then the field has a variable number of elements. Fields representing variable length arrays must always appear at the end of the record definitions. No other fields may appear after the variable array within the record. field dimension may be expressed as formula. The formula parsing rules are as follows:

```
DECIMAL = [0-9]+
OCTAL = 0[0-7]*
HEXADECIMAL = 0x[0-9, a-f, A-F]+
NUMBER = DECIMAL | OCTAL | HEXADECIMAL
VARIABLE = field name
OPERATOR= ‘*’|‘DIV’|‘+’|‘-’|‘bitoffset’|‘bitsize(record name)’
TERM = NUMBER | VARIABLE | TERM OPERATOR TERM | ‘(’ TERM ‘)’

For example, a common dimension formula for a command packet field is:
((command length * 16) - bitoffset) DIV bitsize(field type)
```

field range - This specifies the lowest and highest acceptable value for the field, and is used by the code-generator when producing sanity checks of input arguments or read field values.

field alignment - This specifies the bit-alignment requirement of the field. The code-generator may add pad bits between the previous field and the current field to satisfy the alignment requirement.

variable width - This is the computed number of bits within one element of a variable length array. This width includes any pad bits needed to satisfy alignment requirements of the array elements.

variable offset - This is computed bit offset of the start of a variable length array.

parsed records - This represents the parsed record structure used by the code-generators to produce C++ class definitions and implementations. It consists of an array of keywords and values and has the following form:

```
parsed records = RECORD
TERM = RECORD | FIXED | VARIABLE | FIELD
RECORD = 'record' recordname curoffset TERM* 'endrecord'
FIXED = 'fixed' fieldname curoffset width dimension TERM* 'endfixed'
VARIABLE = 'variable' fieldname curoffset width dimension TERM* 'endvariable'
FIELD = 'field' fieldname curoffset width sign srange erange 'endfield'

where curoffset is the bit-position of the element, width is the number of bits
within the field or 1 element of a fixed or variable length array, dimension is the
formula for the number of elements in the array, sign is 0 if the field is unsigned
and 1 of the field is signed, srange is the lowest value that the field supports,
and erange is the largest value the field supports.
```

21.4 IP&CL Structures Database

21.4.1 File format

NOTE: The IP&CL structures definitions table format is still being design Final definition of the fields is TBD.

The code-generator reads the IP&CL information from an ASCII file. The file contains one line per record field entry, each ending in a newline character. Each entry contains the same number of elements, each separated by a tab character.

21.4.2 Entry Format

Each field entry consists of the following elements, in the listed order:

TABLE 23. IP&CL Field Entry Format

Element	Description
Record Text	This describes the record defining the field
Record Name	This is the name of the record defining the field
Field Number	This is the field position within the record
Subfield Number	This is the subfield position within the record
Owner	This identifies the owner of the record. For ACIS it is always ‘ACIS’
Field Name	This is the name of the field
Mnemonic	This is the field’s 8-character mnemonic. For ACIS software, the mnemonic has the following structure: 1rrffAss, where ‘1’ identifies the mnemonic as an ACIS field, ‘rr’ is the record number in hexadecimal, ‘ff’ is the field number in hexadecimal, ‘A’ specifies the port side, and ‘ss’ is used to identify the type of record being defined. If ‘ss’ is CI, the record is a command packet definition. If it is ‘XF’, it is a telemetry packet, and if it is ‘IT’, it is an interal record.
Data Type	This specifies the data type of the field. This may have the following values: int8, uint8, int16, uint16, int32, uint32, bit, or the name of a record.
Dimension	If the field defines an array, this specifies the number of elements in the array. The dimension statement may consist of a formula, which may refer to fields within the record, may contain parenthesized expressions, and which uses the following operators and keywords: +, -, *, DIV, bitoffset, bitsize(‘record name’)
Maximum Dimension	This specifies the maximum number of elements that can be held by the field.
Variable Record Flag	This indicates whether the fields in the record have constant or variable values
Field Description	This describes the field
Byte Size	This is the number of bytes in 1 element of the field. Its value is computed by TRW.
Total Size	This is the total number of bytes being defined by the field. Its value is computed by TRW.

TABLE 23. IP&CL Field Entry Format

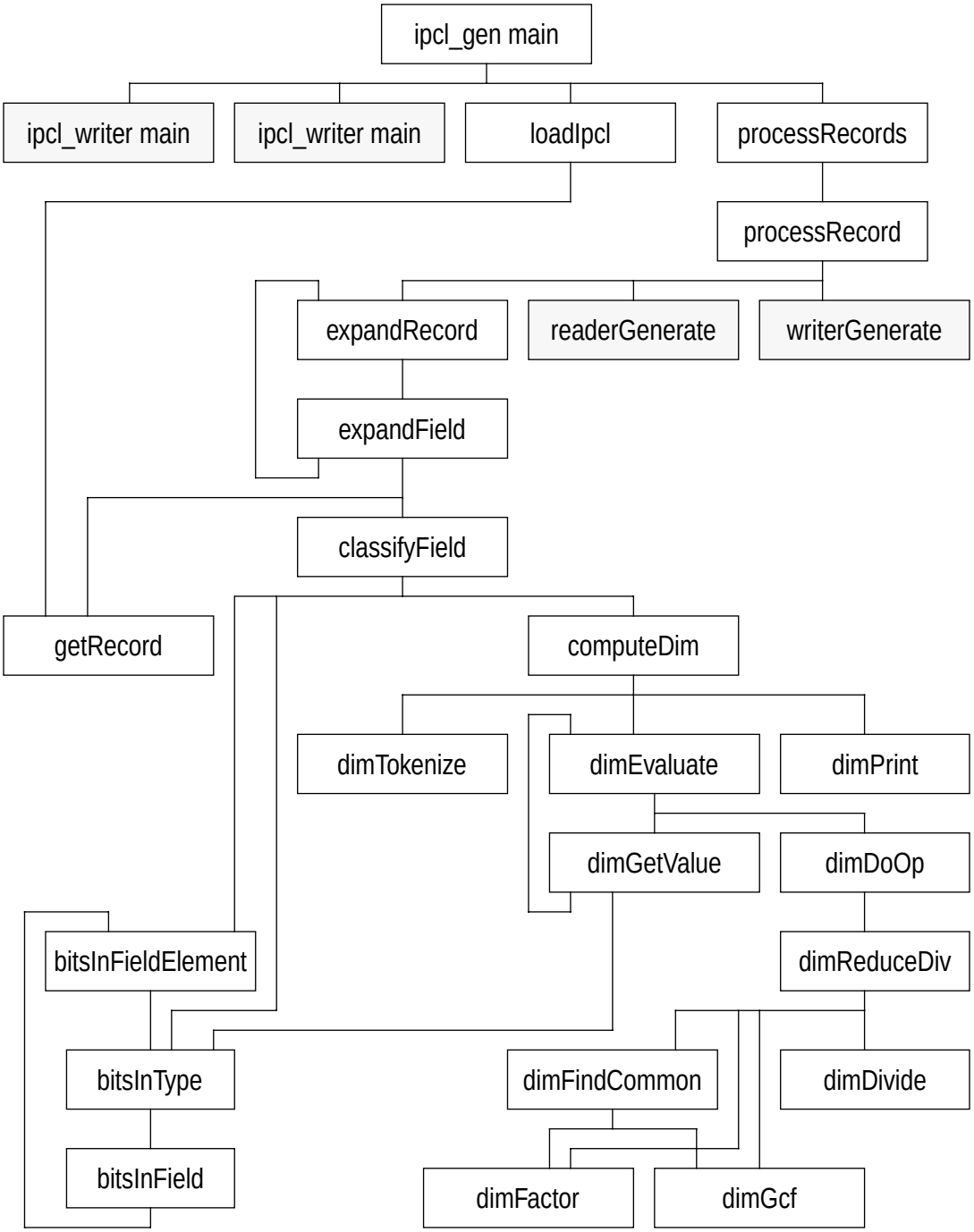
Element	Description
Start Range	This is the minimum value the field can have.
End Range	This is the maximum value the field can have.
Units	This specifies the units of the field
Value	If the field has constant value, this specifies the value.
Format	This identifies the printing format to use for the field
Question	This entry is TBD.
Alignment	This specifies the bit-alignment requirements of the field.
Command Time-out	This specifies the command time-out period, if the field is part of a command packet record definition.

21.5 Script Structure

21.5.1 ipcl_gen.pl

This section illustrates the structure of the main portion of the code-generator script, `ipcl_gen.pl`. The unshaded functions are contained within `ipcl_gen.pl` and the shaded functions are provided by either `ipcl_reader.pl` or `ipcl_writer.pl`.

FIGURE 95. ipcl_gen.pl Structure



- `ipcl_gen main` - This is the main function of `ipcl_gen.pl`. It includes and executes `ipcl_writer.pl` and `ipcl_reader.pl`, calls `loadIpcl`() to load the IP&CL field entries, and calls `processRecords`() to process the loaded entries.
- `ipcl_writer main` - This is the main set of commands executed when `ipcl_writer.pl` is included or invoked. These commands are described more detail below.
- `ipcl_reader main` - This is the main set of commands executed when `ipcl_reader.pl` is included or invoked. This commands are described more detail below.
- `loadIpcl` - This subroutine takes the IP&CL filename as input, opens the file and reads the field entries from the file, parses the entries, using `getRecord`(), to obtain the *`record name`* and *`field index`* contained in the entry, and stores the field entries into the associative array %*`ipcl`*. Upon returning, $*`ipcl`*{*`record name`*} contains the number of fields defined by a particular record, and $*`ipcl`*{*`record name`*, *`field index`*} contains the a copy of the field entry.
- `getRecord` - This function takes one field entry string, consisting of a series of tab-separated elements, and returns an associative array containing each element. The keys used for the array are:

```
rtext - Description of the record
rname - Record name
fnum - Field index
subf - Subfield index
owner - Record owner, always 'ACIS'
fname - Field name
mnem - Field mnemonic
type - Field type
dim - Field dimension
maxdim - Maximum dimension
var - Variable record
ftext - Field description
sizeb - Size of field in bytes
tsize - Total size of field in bytes
srange - Lowest value contained in field
erange - Highest value contained in field
units - Units of value contained in field
value - Fixed value of field
fmt - Printing format for field
quest - Questions???
align - Bit-alignment requirements of field
cmdto - Command Timeout requirements for command records
```

- `processRecords` -This function is responsible for generating C++ classes for all command and telemetry records in the loaded IP&CL. This function iterates through each record definition in %*`ipcl`*, calling `processRecord`() for each.
- `processRecord` - This function is responsible for generating C++ code for a single record. This function takes a *`record name`* as input, and generates C++ code for the passed record. It calls `expandRecord`(), passing the *`record name`* and an initial bit-position of 0, and getting back the *`record type`*, some *`variable length`*

information, and the *parsed record*. It then uses the *record type* to determine if it is generating command packet reader classes, or telemetry packet writer classes, calling readerGenerate() or writerGenerate(), respectively.

- expandRecord - This subroutine takes a *record name* and *current position* as inputs and produces a *parsed record* array, of the form:

  RECORD = 'record' *recordname curoffset* TERM* 'endrecord'

  It starts by placing the 'record' keyword followed by the *record name* and *current position* at the start of its output array. It then gets the number of fields defined by the record from ${*ipcl*{*record name*}, and iterates through each field, calling expandField() for each field it processes. If the field is the first in the record, it sets the expanded field type as the current record type. If the expanded field contained a variable length array, it uses the field's element width and bit-position as the record's variable element width and array bit-position. It then concatenates the parsed record array returned by expandField() to the end of its *parsed record* array. Once all of the fields have been processed, the function appends an 'endrecord' keyword to the *parsed record* array, and returns the *record type*, the new *current position*, the *variable width*, *variable offset*, and the generated *parsed records* array.

- readerGenerate - This function is provided by the script, ipcl_reader.pl. For input, it takes an optional *parent class name*, an optional *basename* to use for the generated classes, an optional *record name* of the packet header to prevent it from generating unnecessary access functions to the header (they're already inherited by parent's functions), an *inline member function flag,* indicating whether or not to generate inline member functions, and a *parsed record* array. readerGenerate creates and emits a C++ *class definition* and *member function implementations* for the record defined in the *parsed records* array. The class provides member functions which read bit-fields from within a command packet buffer.

- writerGenerate - This function is provided by the script, ipcl_writer.pl. For input, it takes an optional *parent class name*, an optional *basename* to use for the generated classes, an optional *record name* of the packet header to prevent it from generating unnecessary access functions to the header (they're already inherited by parent's functions), an optional v*ariable width*, an optional *variable offse*t, a *inline member function flag,* indicating whether or not to generate inline member functions, and a *parsed record* array. readerGenerate creates and emits a C++ *class definition* and *member function implementations* for the record defined in the *parsed records* array. The generated class provides member functions which write bit-fields into a telemetry packet buffer.

- `expandField` - This function is responsible for processing a single field within a record. It takes the *`record name`*, the *`field index,`* and the *`current position`* as input. It retrieves the field entry from $*`ipcl`*{*`record name`*, *`field index`*}, and passes the entry string to `getRecord`() to produce an associative array of the field entries. It then determines the field type by examining the field's *`mnemonic`*. If the last two characters of the *`mnemonic`* are 'CI', then it sets its local *`record type`* to a command packet. If the last two characters are 'XF', it sets *`record type`* to a telemetry packet, otherwise, it assumes the *`record type`* is internal. It then check's the field's alignment requirements, and adds any needed padding to the current position. It then passes the *`record name`*, the *`field index`*, the current offset and the associative array of *`field elements`* to `classifyField`() to compute and determine various properties of the field. `classifyField`() returns and associative array of *`field properties`*, *`%props`*. The function then proceeds to produce append information to the parsed records array. If the field defines a fixed length array ($*`props`*{*`isfixedarray`*}), the function appends the keyword 'fixed', followed by the *`field name`*, the *`current offset`*, the *`width`* of one element in the array ($*`props`*{*`arraywidth`*}), and the *`dimension`* of the array ($*`props`*{*`limit`*}) to the end of the *`parsed records`* array:

  ```
  FIXED = 'fixed' fieldname curoffset width dimension TERM* 'endfixed'
  ```

  Otherwise, if the field is a variable length array ($*`props`*{*`isvararray`*}) the function sets the *`variable width`* to return to the element's width ($*`props`*{*`arraywidth`*}, and the *`variable offset`* to the *`current position`*. It then appends the keyword 'variable', followed by the *`field name`*, the *`current offset`*, the *`width`* of one element in the array ($*`props`*{*`arraywidth`*}), and the *`dimension`* formula for the array ($*`props`*{*`limit`*}) to the end of the *`parsed records`* array:

  ```
  VARIABLE = 'variable' fieldname curoffset width dimension TERM* 'endvariable'
  ```

  If the field has a simple data type ($*`props`*{*`basetype`*}), the function appends the keyword 'field', followed by the *`field name`*, the *`current offset`*, the *`width`* in bits, the *`sign`*, the field's *`range`*, and the keyword 'endfield' to the *`parsed records`* array and advances the current offset by the bit-width of the field ($*`props`*{*`bitwidth`*}):

  ```
  FIELD = 'field' fieldname curoffset width sign srange erange 'endfield'
  ```

  Otherwise, the field has a complex data type and the function recurses and passes the *`field type`* and *`current offset`* to `expandRecord`() to expand the type into a *`parsed record`* array, which is then concatenated to its local array. If the processed record ends with a variable length array, the function logs the record's variable width and offset.

  If the processed field was a fixed length or variable length array, the function respectively appends the 'endfixed' or 'endvariable' keyword to the parsed records array.

  It then returns with the *`record type`*, the modified *`current offset`*, the *`variable width`*, *`variable offset`*, and the modified *`parsed record`* array.

- `classifyField` - This function determines various properties of a particular field. For input, it takes the *`record name`*, the *`field index`*, the *`current position`*, and the associative array of *`field elements`*. On output, it returns an associative array, where the keys are as follows:

```
basetype - Indicates if field type refers to an integer or bit-field
isvararray- Indicates if field defines a variable length array
isfixedarray - Indicates if field defines a fixed length array
bitwidth - For integer/bit types, contains the bit-width of the field
limit - For arrays, defines the number of elements in the array
sign - For integer types, indicates if field is unsigned or signed
srange - For integer/bit types, indicates lowest value of field
erange - For integer/bit types, indicates largest value of field
arraywidth - For arrays, contains bit-width of 1 element in the array
```

  The function starts by passing the field's dimension expression, current offset, and alignment requirements to `computeDim`() to reduce the dimension expression to its simplest form, and assigns the result to $*`props`*{*`limit`*}. If the expression is not 1, then the field defines an array, and the function checks to see if the expression represents a single number, or represents a formula. If the former, then the array is fixed length, and the function sets $*`props`*{*`isfixedarray`*}. If the expression is not a simple number, then the field represents a variable length array, and the function sets $*`props`*{*`isvararray`*}.

  It then checks the field type against the base integer and bit type keywords. If the type matches, the function sets $*`props`*{*`basetype`*}, and calls *`bitsInType`*() to extract the number of bits in the field and copying the result to $*`props`*{*`bitwidth`*}. If the field's type is a bit-field, and has a dimension less than or equal to 32, it converts the fixed length array of bits into a single element by setting $*`props`*{*`bitwidth`*} to $*`props`*{*`limit`*}, and setting the $*`props`*{*`limit`*} to 1. The function then determines the sign of the field, and sets $*`props`*{*`sign`*} to 0 if the field is unsigned, and to 1 if it is signed. It then assigns the start and end range properties of the field based on the passed entry's value, srange, and erange components.

  Finally, if the field is a fixed length or variable length array, the function calls `bitsInFieldElement`() to compute the size of a single element in the array, and adjusts the result by any alignment requirements of the field. The result is stored in $*`props`*{*`arraywidth`*}.

  The function then returns the constructed associative array of properties.

- *`bitsInType`* - This function computes the number of bits within a data *`type`*. The inputs to the function are the *`type name`*, and the *`current offset`*. If the *`type`* is a bit, it returns 1 for the width. If it is a simple integer type, it returns the appropriate number of bits for that type. If the *`type`* is a *`record name`*, it iterates through each field in the record, passing the *`record name`*, f*`ield index`*, and *`current position`* to `bitsInField`() for each field in the record, and adds the returned bit-size to the current total for the record and to the *`current position`*. Once all fields have been processed, it returns the computed total.

- `bitsInFieldElement` - This function computes the number of bits within 1 array element of a field, or if the field is not within an array, then of the field itself. The inputs to the function are the *`field entry`* to be processed, and the *`current position`*.

It returns the computed size of the field element. The function passes the field entry string to `getRecord`() to get an associative array of the entries elements. It then determines if any padding is needed to maintain the alignment of the field, based on the *`current position`* and the *`alignment`* requirements of the field. If so, it adds the needed padding to the total size of the field element and the current position. The function then calls `bitsInType`() to compute the size of 1 element of the field, and returns the padded size.

- `bitsInField` - This function computes the total number of bits within all elements of a field. This function cannot be used on a field which defines or contains variable length array. The inputs to the function are the `field entry` string and the `current position`. The function calls `getRecord`() to obtain an associative array of the field entry elements. It then calls `bitsInFieldElement`() to determine the size of a single element in the field (if the field is an array). It then multiplies the returned size with the *`dimension`* of the field and returns the result.
- `computeDim` - This function interprets and reduces a field's dimension expression to its simplest form. Its input arguments are the field's *`dimension`* expression string, the *`current position`* of the field, and the *`field alignment`* requirement. The function returns a reduced form of the expression. The function starts by checking if the *`dimension`* is already a simple number. If it is, the function just returns the *`dimension`* string. If not, it passes the *`dimension`* string to `dimTokenize`(), which splits the string into tokens and returns the array of tokens. The function then passes the *`current position`*, the field's *`alignment`* requirement and the token array to `dimEvaluate`(), which interprets and simplifies the expression. `dimEvaluate`() returns an array of token strings. From the top level, the array consists of 1 item, which is a tab-separated of expression elements. The function splits this string into an array, clears the variable $*`dimStrOut`*, and calls `dimPrint`() to reformat the tokens into an expression string, storing the result into $*`dimStrOut`*. The function then returns the formed string.
- `dimTokenize` - This function parses a field's *`dimension`* string into an array of tokens. The input to the function is a field's *`dimension`* string, and it returns an array of tokens. The function first builds a regular expression string which consist of an "or" pattern of expressions, where each matches a particular token. It then splits the *`dimension`* string into an array using the regular expression. It then enters a loop which re-constructs any multiple word symbol tokens back into a single token string. It then returns the token array.

- `dimEvaluate` - This function reduces a list of dimension tokens into a simplified set of dimension expression tokens. The input to the function is the *`current position`*, the *`alignment`* requirement of the field, and an array of *`dimension`* statement tokens. The routine returns a reduced set of tokens. This routine recurses, and once it returns to the top level, the array of tokens will contain 1 element, a string of tab-separated dimension expression statements. The string contains the simplified formula, consisting of tab-separated operators and values in Reverse-Polish-Notation (RPN). Variables names within the array are surrounded by '#'s. This eases searching and replacing the variable names with source-code names. For example, the formula:

  ```
  ((length * 16) - 11) / 9
  ```

  is expressed as (parenthesis are shown only for readability):

  ```
  / - * #length# 16 11 9
  ```

  A single symbol or value, with no operators, means that the dimension expression reduced to a single value. An expression starting with the operators '*', '/', '+', '-' are followed by a left and right side expression.

- `dimGetValue` - This function process the next value token from the array of *`dimension`* expression tokens. The input to the function is the *`current position`*, the field *`alignment`* requirements, and the current token array. The function returns the token array, with the resolved argument. If the first token is an open parenthesis, the function recursively calls `dimEvaluate`() to resolve the expression contained within the parenthesis, replacing the expression tokens with the resolved expression. If the first token is a minus sign, the function forms a binary expression by inserting 0 to the beginning of the token array. If the first token is a hexadecimal number, the function replaces the token with the decimal equivalent. If the first token is the 'bitoffset' keyword, the function replaces the keyword with the passed current position. If the first token is the 'bitsize(*`record name`*)' expression, the function calls `bitsInType`() to compute the number of bits in the specified record, adjusts the result by the passed field alignment requirement, and replaces the expression with the computed size. If the operator is a symbol, the function places '#' characters on either side of the symbol allow code-generation to locate and replace the symbol within an expression string.

- `dimDoOp` - This function attempts to evaluate an expression. The inputs to the function are the left side of the expression, the operator to use, the right side of the expression, and the current array of dimension tokens. On output, the result is inserted into the beginning of the token array and the array is returned to its caller. The function first checks the left and right side expressions. If either is not a simple numeric value, the function forms an RPN expression of the form 'operator left right', using `dimReduceDiv`() to reduce division expressions if the operator is 'DIV' and the right argument is a number. If both sides of the expression are numbers, the function evaluates the expression.

- `dimReduceDiv` - This function attempts to reduce a division expression to a simpler form. The function takes the denominator value, followed by the array of dimension tokens containing the numerator as input. On output, the function returns an array of tokens containing the reduced expression. The function calls `dimFindCommon`() to find some common factors between all terms in the numerator. It then calls `dimFactor`() to partially factor the denominator value. It then calls `dimGcf`() to find

the common components between the terms in the numerator and in the denominator. The function then multiplies the common components to form a common factor, and divides each term in the numerator by this factor. The result is inserted into the token array. It then checks the denominator. If the denominator and the factor are not the same value, the function inserts the '/' operator into the array, and appends the result of 'den / factor' to the end of the array.

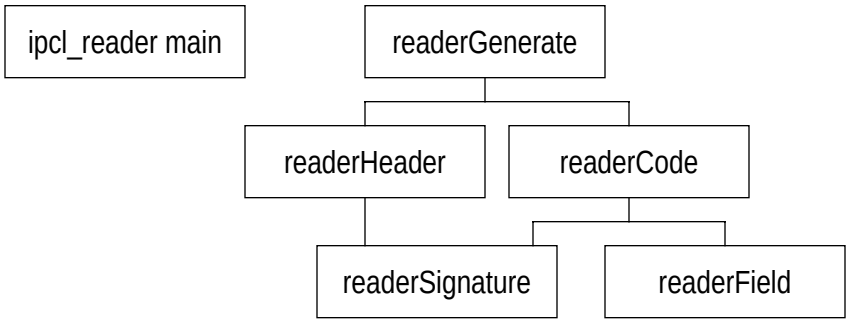
- `dimFactor` - This function returns a string containing a list of factors of a particular number. The function takes the number to factor for input, and returns a string of $; separated factors of the value. The factoring may be incomplete, and the last value in the list contains the un-factored remainder. The product of all elements in the list result in the input value. If the input value is 0, the function returns '1'. If not, it attempts to factor the number. It forms a small list of prime numbers, and iterates through the list. While the current value is evenly divisible by a given prime, the function adds the prime to the output list and divides the value by the prime. Once the value reaches '1' or the list of primes is exhausted, the function sorts the resulting list into descending order, forms the output string from the list and returns the string.
- `dimFindCommon` - This function finds common factors in an expression's terms. The function takes an array of expression tokens on input, outputs a list of factors in the $*dimFactors* string, and returns the surviving expression tokens to its caller. The function removes the first value from the token array. If it is a '+' or '-', the function finds the common factors between the left and right sides of the expression. If it is a '*', it factors both sides of the expression and combines the two into a single list. If the value is a number, it factors the number.
- `dimGcf` - This function compares two lists of factors, and produces a list of those in common between the two lists. The input is two strings, containing factor values delimited by $;. On output, the function returns a string containing the common values delimited by $;. The function clears its output array, splits the second input string into an array, and retrieves the first value from the array. It splits the first string into an array and iterates through the array. For each element in the first array, the function searches the second array until it finds a value less than or equal to the value from the first. If the values are equal, the function adds the value to the output list, and advances both lists. Once all values have been check, the function joins the output array into a single string and returns it to its caller.
- `dimDivide` - This function divides a compound expression by a single value. On input, the function takes the denominator value to use, and an array of tokens containing the numerator expression. The function assumes that the denominator divides evenly into all elements of the numerator. The function extracts the first element from the numerator expression. If it is a '+' or '-', the function recurses to divide each side of the expression by the denominator. If it is a '*', the function recurses to try to divide the left hand side of the expression by the denominator, and if it fails, it divides the right side of the expression. If the value is a '/', the function recurses to divide the numerator of the sub-expression by the passed denominator. If the value is a number, the function explicitly evaluates the expression. The function returns the array of divided expression tokens to its caller.

- `dimPrint` - This function converts a reduced dimension expression, consisting of an array of expression operators, values and variables in RPN (see `dimEvaluate`()), into C/C++ form, except for variable names, which are not expanded into source-code names at this stage. The variable name expansions are handled in the code-generation stage. The input to the function is the array of operator, values and variables, and the result string is placed in the variable $*dimStrOut*. The caller is responsible for clearing $*dimStrOut* prior to calling this function. This function de-queues the first element from the token array and compares it to known operators. If it is a '+', '-' or '/', the function writes the form '(left expression operator right expression)' to $*dimStrOut*, recursively calling `dimPrint`() to print the left and right side expressions. If the first element is a '*', the function checks the value of left and right side arguments, if either is '1', then it writes the other argument's value. If neither argument is '1', the function writes the form '(left expression * right expression)', recursing to expand the left and right expressions.

21.5.2 ipcl_reader.pl

This section illustrates the structure of the command packet reader portion of the code-generator script, `ipcl_reader.pl`.

FIGURE 96. ipcl_reader.pl structure



- `ipcl_reader main` - This function initializes the global variables used by the `ipcl_reader.pl` **Perl** script. The function initializes the following configuration associative array parameters:

  $*cfg*{*inputbits*} - Number of bits in command buffer word = 16
  $*cfg*{*inputname*} - Variable name of command buffer "inputbuf"

  It also initializes the array of masks used by the reader code, $*readerMasks*[]. Each element of the array contains a right-justified number of '1's corresponding to the element's index in the array.

- `readerGenerate` - This function generates a C++ class for the passed command packet record definition. For input, it takes an optional *parent class name*, an optional *basename* to use for the generated classes, an optional *record name* of the packet header to prevent it from generating unnecessary access functions to the header (they're already inherited by parent's functions), an optional v*ariable width*, an optional *variable offse*t, a *inline member function flag,* indicating

whether or not to generate inline member functions, and a *parsed record* array. `readerGenerate` creates and emits a C++ *class definition* and *member function implementations* for the record defined in the *parsed records* array. The function forms the class name from the record name by replacing any spaces with underscores '_', and appending the result to the passed *basename*. It then appends the ".H" extension to the class name and creates the include file for the class using the formed name. It then writes a standard header to the include file, and include's its parent class's include file. The function then calls `readerHeader`() to write the class definition to the include file, and closes the file. The function then checks if inline functions were requested. If so, it re-opens the file for appending. If not, it creates the class's implementation file by appending the ".C" extension to the classname, writes a standard header to the file, and writes an include directive to the class's include file. The function then calls `readerCode`() to write the implementation to the open file (either the include file for inline's otherwise the new source file). Once the implementation has been written, the function closes the open file.

- `readerHeader` - This function writes the class definition for a command packet class. The inputs to the function include the *parent class name*, the command's *class name*, the *record name* of the packet header, and the *parsed record* array. The function first prints the class declaration:

  ```
  class class_name : public parent name
  {
  public:
  ```

  It then iterates through each tokens in the *parsed record* array.

  If the token is the 'record' keyword, the function increments a nested record count, extracts the record name and compares it with the passed header record name. If they match, the function sets a skip record counter. If not, the function prints a comment indicating that it is expanding a nested record definition.

  If the token is an 'endrecord' keyword, the function decrements the nested record and skip record counters.

  If the token is a 'fixed' keyword, the function increments an array flag counter.

  If the token is an 'endfixed' keyword, the function decrements the array flag counter.

  If the token is a 'variable' keyword, the function retrieves the field name of the variable length array, and saves the name until all of the tokens have been processed. It then increments the array flag counter.

  If the token is an 'endvariable' keyword, the function decrements the array flag counter.

  If the token is a 'field' keyword, the function retrieves the field name and sign. If the skip record counter is zero, the function calls `readerSignature`() to emit the member function access function declaration for the field. If the skip record counter is not zero, then an accessor function for the field is not produced. The function then checks and consumes the 'endfield' keyword.

  Once all of the tokens have been processed, the function checks the variable length field name. If one was defined, the function emits the signature for a function which returns the number of elements in the variable length array:

  ```
  unsigned get_CountOf_fieldname() const;
  ```

The function then closes the class definition with a closing brace and semicolon.

- `readerCode` - This function implements the command packet accessor functions for each field of the command record. The inputs to the function include the *`parent class name`*, the command's *`class name`*, the *`record name`* of the packet header, whether or not to produce *`inline`* functions, and the *`parsed record`* array. The function iterates through each tokens in the *`parsed record`* array.

  If the token is the 'record' keyword, the function increments a nested record count, extracts the record name and compares it with the passed header record name. If they match, the function sets a skip record counter. If not, the function prints a comment indicating that it is expanding a nested record definition.

  If the token is an 'endrecord' keyword, the function decrements the nested record and skip record counters.

  If the token is a 'fixed' keyword, the function increments an array flag counter, and sets local array offset, array width and array limit variables. The symbols within the array limit expression are converted into get_fieldname() functions. For example:

  ```
  ((#field_name# * 16) - 32)/14
  ```

  is converted to:

  ```
  ((get_field_name() * 16) - 32)/14
  ```

  If the token is an 'endfixed' keyword, the function decrements the array flag counter.

  If the token is a 'variable' keyword, the function retrieves the field name of the variable length array, array offset, array width and array limit. It then increments the array flag counter. The array limit expression is converted as described above for the 'fixed' keyword.

  If the token is an 'endvariable' keyword, the function decrements the array flag counter.

  If the token is a 'field' keyword, the function retrieves the field name, bit offset, bit width, sign, starting value and ending value. The function then checks and consumes the 'endfield' keyword. If the skip record counter is zero, the function calls `readerSignature`() to emit the member function access function declaration for the field. It then calls `readerField`() to emit the implementation of the function. If the skip record counter is not zero, then an accessor function for the field is not produced.

  Once all of the tokens have been processed, the function checks the variable length field name. If one was defined, the function implements the function which returns the number of elements in the variable length array:

  ```
  unsigned classname::get_CountOf_fieldname() const
  {
      return array limit expression;
  };
  ```

- `readerSignature` - This function writes the signature of a field's accessor member function. The inputs to this function are the class name, the field name, the sign of the field, whether or not the field is contained within an array, whether or not the function is to be implemented as an inline function, and whether to emit the signature as part of a class declaration, or as part of the implementation of the function. The name of the generated accessor is "get_field_name." If the field is signed, the accessor returns an **`int`**

result. If the field is unsigned, it returns an **`unsigned`** value. If the field is within an array, the generated accessor takes a **`unsigned`** *`index`* argument. If the signature is to be part of the implementation of the function, the class name is prepended to the accessor name (i.e. classname::get_field_name). For example, the signature for an unsigned array element within a class declaration is:

```
unsigned get_field_name(unsigned index) const;
```

Whereas, at the start of its implementation, it looks like:

```
unsigned record_name::get_field_name (unsigned index) const
```

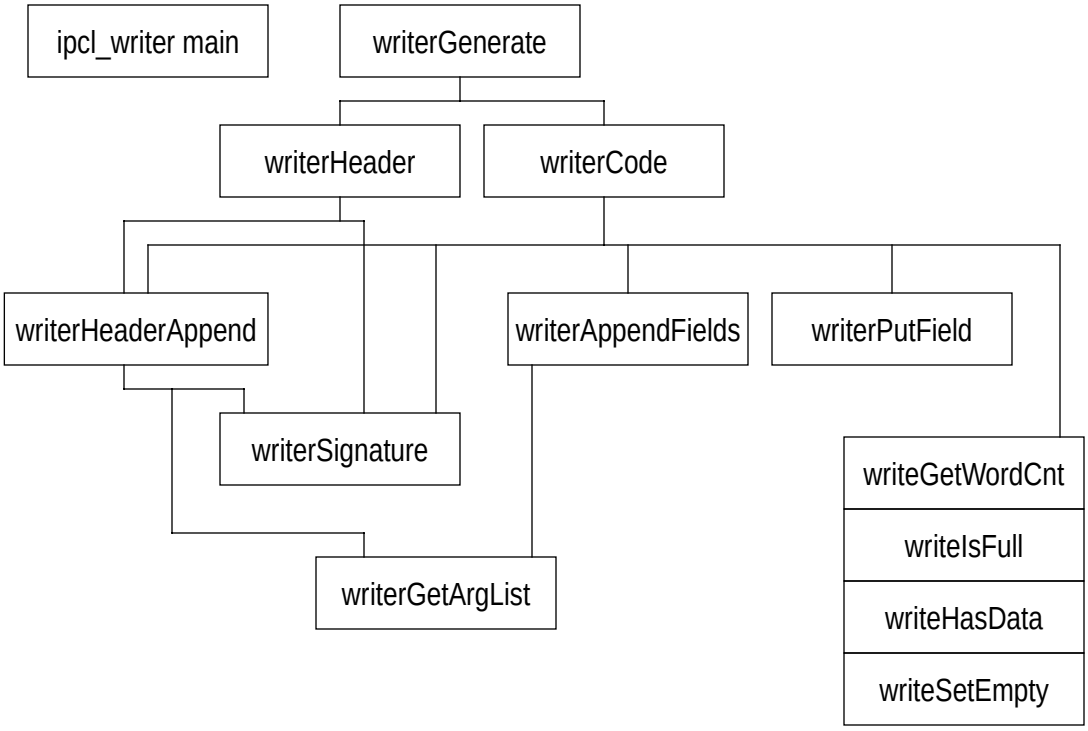
- `readerField` - This function implements the body of a field accessor. The inputs to the function are the name of the field, the bit-offset of the field, relative to the start of the record, the number of bits in the field, its sign, start value, end value, whether or not it is part of an array, and if so, the bit-offset to the start of the array, the number of bits in each array element, and the formula for the number of entries in the array. This function starts by writing an opening brace. If the field is in array, the function writes an assertion on the range of the input index. It then prints a comment indicating the position, and width of the field, and if it is within an array, the array element width. It then writes code which calls the inherited function, ‘getField()’. If the field is signed, it then emits code which sign-extends the result by a combination of shifts and type-casts. It then emits asserts which check the range of the value. Finally, the function writes code to return the value and writes the closing brace. For example, given a signed 3-bit field, starting at bit-position 5, whose range is from 1 to 5, and contained in an array of elements 17 bits wide, and containing at most 10 elements, the implementation might look as follows:

```
{
    // ---- Range check index argument ----
    assert (index < 10)

    // ---- Offset = 5, Width = 3, Structure = 17----
    unsigned out = getField (5, 3, 0x7, index, 17);
    out = unsigned(int(out << 30) >> 30);

    // ---- Value range checks ----
    assert (out >= 1);
    assert (out <= 5);

    return int(out);
};
```

21.5.3 ipcl_writer.pl

This section illustrates the structure of the telemetry packet writer portion of the code-generator script, `ipcl_writer.pl`.

FIGURE 97. ipcl_writer.pl structure



- `ipcl_writer main` - This function initializes the global variables used by the `ipcl_writer.pl` **Perl** script. The function initializes the following configuration associative array parameters:

  ```
  $cfg{outputbits} - # of bits in word of the telemetry buffer = 32
  $cfg{ouptputname} - Variable name of telemetry buffer “outputbuf”
  $cfg{buflimit} - Max. words in telemetry buffer = 1024 * 32
  ```

  It also initializes the array of masks used by the writer code, $*writeMasks*[]. Each element of the array contains a right-justified number of ‘1’s corresponding to the element’s index in the array.

- `writerGenerate` - This function produces a C++ class which writes a record’s fields into a telemetry packet buffer. For input, it takes an optional *parent class name*, an optional *basename* to use for the generated classes, an optional *record name* of the packet header to prevent it from generating unnecessary access functions to the header (they’re already inherited by parent’s functions), an optional v*ariable width*, an optional *variable offse*t, a *inline member function flag,* indicating whether or not to generate inline member functions, and a *parsed record* array. `writerGenerate` creates and emits a C++ *class definition* and *member function implementations* for the record defined in the *parsed*

records array. The function forms the class name from the record name by replacing any spaces with underscores '_', and appending the result to the passed *basename*. It then appends the ".H" extension to the class name and creates the include file for the class using the formed name. It then writes a standard header to the include file, and include's its parent class's include file. The function then calls `writerHeader`() to write the class definition to the include file, and closes the file. The function then checks if inline functions were requested. If so, it re-opens the file for appending. If not, it creates the class's implementation file by appending the ".C" extension to the classname, writes a standard header to the file, and writes an include directive to the class's include file. The function then calls `writerCode`() to write the implementation to the open file (either the include file for inline's otherwise the new source file). Once the implementation has been written, the function closes the open file.

- `writerHeader` - This function writes the class definition for the telemetry packet writer class. The inputs to the function include the *parent class name*, the telemetry packet's *class name*, the *record name* of the packet header, and the *parsed record* array. The function first prints the class declaration:

```
class class_name : public parent name
{
public:
```

  It then iterates through each tokens in the *parsed record* array.

  If the token is the 'record' keyword, the function increments a nested record count, extracts the record name and compares it with the passed header record name. If they match, the function sets a skip record counter. If not, the function prints a comment indicating that it is expanding a nested record definition.

  If the token is an 'endrecord' keyword, the function decrements the nested record and skip record counters.

  If the token is a 'fixed' keyword, the function increments an array flag counter.

  If the token is an 'endfixed' keyword, the function decrements the array flag counter.

  If the token is a 'variable' keyword, the function retrieves the field name of the variable length array, and saves the name until all of the tokens have been processed. It then increments the array flag counter. The function then calls `writerHeaderAppend`() to form the signature for the access function which writes each element of the field and consumes all fields until an 'endvariable' keyword is reached.

  If the token is an 'endvariable' keyword, the function decrements the array flag counter.

  If the token is a 'field' keyword, the function retrieves the field name and sign. If the skip record counter is zero, the function calls `writerSignature`() to emit the member function access function declaration for the field. If the skip record counter is not zero, then an accessor function for the field is not produced. The function then checks and consumes the 'endfield' keyword.

  Once all of the tokens have been processed, the function emits the following signature:

```
unsigned getWordCount() const - Get number of words in packet buffer
```

  It then checks the variable length field name. If one was defined, the function emits the signatures for the following functions:

```
void setEmpty() - Set the number of appended items in the packet to 0
Boolean isFull() const - Determine if packet buffer is full
```

```
Boolean hasData() const - Determine if packet has any appended data
```

The function then emits a protected instance variable, ‘unsigned _appended;’, which holds the number of elements appended to the end of the telemetry packet buffer.

The function then closes the class definition with a closing brace and semicolon.

- `writerCode` - This function implements the member functions of the telemetry packet writer class. The inputs to the function include the *`parent class name`*, the command’s *`class name`*, the *`record name`* of the packet header, whether or not to produce *`inline`* functions, and the *`parsed record`* array. The function iterates through each tokens in the *`parsed record`* array.

  If the token is the ‘record’ keyword, the function increments a nested record count, extracts the record name and compares it with the passed header record name. If they match, the function sets a skip record counter. If not, the function prints a comment indicating that it is expanding a nested record definition.

  If the token is an ‘endrecord’ keyword, the function decrements the nested record and skip record counters.

  If the token is a ‘fixed’ keyword, the function increments an array flag counter, and sets local array offset, array width and array limit variables.

  If the token is an ‘endfixed’ keyword, the function decrements the array flag counter.

  If the token is a ‘variable’ keyword, the function retrieves the field name of the variable length array, array offset, array width and array limit. It then increments the array flag counter. The function calls `writerHeaderAppend`() to print the declaration of the writer function for the array elements, and then `writerAppendFields`() to emit the function implementation, and consume the tokens up to the ‘endvariable’ keyword.

  If the token is an ‘endvariable’ keyword, the function decrements the array flag counter.

  If the token is a ‘field’ keyword, the function retrieves the field name, bit offset, bit width, sign, starting value and ending value. The function then checks and consumes the ‘endfield’ keyword. If the skip record counter is zero, the function calls `writerSignature`() to emit the member function access function declaration for the field. It then calls `writerPutField`() to emit the implementation of the function. If the skip record counter is not zero, then an accessor function for the field is not produced.

  Once all of the tokens have been processed, the function checks the variable length field name. If one was defined, the function calls `writeGetWordCnt`() to implement the function which returns the number of words in the telemetry packet buffer, `writeSetEmpty`() to emit the function which clears the ‘_appended’ instance variable, `writeIsFull`() to implement the function which determines if the telemetry packet buffer is full, and `writeHasData`() to implement the function which determines if the telemetry packet buffer has data. If the packet is not variable length, then the function calls `writeGetWordCnt`() to implement the function which returns the static size of the telemetry packet.

- `writerSignature` - This function writes the signature of a field’s accessor member function. The inputs to this function are the class name, the field name, whether or not the field is contained within an array, whether or not the function is to be implemented

as an inline function, whether to emit the signature as part of a class declaration, or as part of the implementation of the function, and an array of function arguments. If the field does not define a variable length array, the name of the generated accessor is "put_field_name." If the field defines a variable length array, the name of the generated accessor is "append_field_name." If the field is within a fixed length array, this function inserts a **unsigned** *index* argument to the front of the signature's argument list. If the signature is to be part of the implementation of the function, the class name is prepended to the accessor name (i.e. classname::put/append_field_name). For example, the signature for an variable array element containing one subfield, within a class declaration is:

```
void append_field_name(unsigned index, unsigned subfield);
```

Whereas, at the start of its implementation, it looks like:

```
void record_name::append_field_name (unsigned index, unsigned subfield)
```

- `writerHeaderAppend` - This function writes the signature for the accessor function for a field which defines a variable length array. The inputs to this function are the class name, the field name, whether or not the signature is part of the class definition, or part of the implementation, and the array of *`parsed record`* tokens. This function returns *`parsed record`* tokens after and including the 'endvariable' keyword. The function calls `writerGetArgList`() form a $;-separated list of fields within the variable array element, and to consume all of the *`parsed record`* tokens up to the 'endvariable' keyword. This function then splits the argument string into an array, and extracts the argument type and name, forming an array of argument statements. The function then calls `writerSignature`() to emit the signature for the append function. It then returns the reduced *`parsed record`* token array.
- `writerPutField` - This function implements an accessor function which writes a bit-field into the telemetry packet buffer. The inputs to the function include the class name, the field name, the bit-offset of the field, the number of bits in the field, whether or not the field is within a fixed-length array, and if so, the number of bits in one element of the array, the number of elements in the array. The start and end values for the field are also passed as input. This function writes the body of the access function, starting with the opening brace. If the field is part of an array, it writes an assert to check the passed index against the number of elements in the array. It then writes asserts to check the input value against the field's starting and ending limits. This function then writes a comment indicating the bit-position, width and, if applicable, array element width. The function then emits a call to the inherited function 'putField().' Finally, the function emits a return statement, followed by a closing brace. For example, given a signed 3-bit field, starting at bit-position 5, whose range is from 1 to 5, and contained in an array of elements 17 bits wide, and containing 10 elements, the implementation might look as follows:

```
{
    // ---- Range check index argument ----
    assert (index < 10)
```

```
        // ---- Value range checks ----
        assert (input >= 1);
        assert (input <= 5);

        // ---- Offset = 5, Width = 3, Structure = 17----
        putField (input, 5, 3, 0x7, index, 17);

        return;
    };
```

- `writerAppendFields` - This function implements an accessor function which appends the subfields from a variable length array element onto the end of the telemetry packet buffer. The inputs to the function include the class name, the field name defining the array, the bit-offset of the array, the number of bits within one element of the array, the maximum number of elements that the array can hold, and the parsed record array. The function returns the reduced parsed record token array. The function first calls `writerGetArgList`() to produce a string containing $;-separated list of field statements, and to consume the field tokens from the parsed record array. The function then starts writing the body of the function, starting with the opening brace. It then emits an assertion which checks if the telemetry buffer is full. The function then iterates through the argument list. For each argument, the function extracts the subfield's name, count if it is part of a subarray, bit offset, bit width, sign and range. It then iterates through the subarray's elements, emitting an assertion for the argument's value, and emitting a call to the inherited function 'appendField().' Once all of the arguments have been processed, the function emits an operation to advance the '_appended' instance variable, and finally emits a return statement followed by a closing brace. For example, consider a field which defines variable length array of elements, whose subfields are 'sub 1' and 'sub2.' 'sub 1' is 10-bits wide and starts 12-bits into the packet. 'sub 2' is 4-bits wide, and starts 22-bits into the packet. Assume that no ranges were specified for the fields. The generated code may look as follows:

```
    {
        // ---- Range check index argument ----
        assert (isFull() == BoolFalse);

        // ---- sub 1 :: Offset = 12, Width = 10 ----
        appendField (sub_1, 12, 10, 0x3ff, _appended, 26);

        // ---- sub 2 :: Offset = 22, Width = 4 ----
        appendField (sub_2, 4, 22, 0x0f, _appended, 26);

        return;
    };
```

- `writerGetArgList` - This function processes a field defining a variable length array. It extracts the series of subfields defined for the defining field. The input to the function is the array of *`parsed records`*. On output, the function returns a $;-delimited string of subfield specifiers, followed by the remaining array of *`parsed records`*. The format for the subfield specifiers consist of an argument specifier string followed by a set of field properties surrounded by angle brackets:

```
    argument string<field count offset width sign srange erange>
```

The function iterates through each of the *parsed record* tokens until it reaches an ‘endvariable’ keyword.

The function skips over ‘record’ and ‘endrecord’ delimited sections. When it encounters a ‘field’ keyword, the function extracts the field name, offset, width, sign, and range. It then checks and removes the ‘endfield’ keywords. The function then tests the field name. If the name consists of a basename followed by a 0, then it test subsequent fields to see if the fields can be combined into an array argument. If the subsequent field name consists of the same basename, followed by a 1, and its bit-width and sign match the first argument, the function iterates over subsequent fields until either the basename, the sequence number, bit-width or sign pattern are broken. If the subfield is not part of an array, the function emits an **unsigned** or **int** argument expression, followed by the field’s properties. If the subfield is part of an array, it emits one argument expression for all of the participating fields, consisting of:

```
const unsigned or int basename[field count]
```

Once all of the subfields have been processed, the function forms and returns the $;-delimited argument list string.

- writeGetWordCnt - This function emits an implementation of an accessor function which returns the number of words contained within a telemetry packet buffer. The inputs to the function are the class name, the width of one element of a variable length array, and the bit-position of the end of the fixed length portion of the packet. If no variable length array is present in the telemetry packet, the caller passes 0 for the array width. The function starts by writing the signature of the generated code. It then emits a comment indicating the position of the end of the fixed-length portion of the buffer, and the size of one element of the variable length array. If a variable length record is being processed, the function emits code to compute the number of words in the array based on the value of the ‘_appended’ instance variable. If not, the function emits code to compute the length based on constants. For example, the generated code may look as follows:

```
unsigned class_name::getWordCount (void) const
{
    // ---- Offset 10, Element Size 20 ----
    unsigned wordcnt = 10 + (_appended * 20);
    wordcnt = (wordcnt + 31) / 32;
    return wordcnt;
};
```

- writeIsFull - This function emits code which determines whether or no the telemetry packet is full. This code is only emitted for variable length record definitions. The inputs to the function are the class name, the number of bits in one array element, and the bit-position to the start of the variable length array. The function emits code which computes the number of words in the array given one more element, and returns whether or not the result exceeds the limit for the packet. The function starts by emitting the function signature and opening brace. If the widths and offsets are evenly divisible by the number of bits in an output word, the function divides each element by the output bit-width. The function then emits code to compute the length of the packet, given 1 more element, and compares it to the limit supported by the packet buffer. For example, given an element of 10 and an offset of 11, the generated code may look as follows:

```
Boolean class_name::isFull (void) const
{
    // ---- Offset 11, Element Size 10 ----
    unsigned offset = 11 + ((_appended + 1) * 10);
    return (offset > 4096) ? BoolTrue : BoolFalse;
};
```

- `writeHasData` - This function emits code which determines if a telemetry packet has had any data appended to it. This code is only emitted for records defining variable length telemetry packets. The input to this function is the class name. This function emits the following code:

```
Boolean class_name::hasData (void) const
{
    return (_appended != 0) ? BoolTrue : BoolFalse;
};
```

- `writeSetEmpty` - This function emits code which zeros the counter which tracks the number of element appended to a telemetry packet buffer. This function is emitted only for variable length telemetry packets. The code emitted by this function is as follows:

```
void class_name::setEmpty (void)
{
    _appended = 0;
};
```

22.0 Command/Parameter Reader Templates (36-53232.02 01)

22.1 Purpose

The purpose of the collection of command and parameter block format reader classes is to extract bit-packed fields from buffers containing commands or parameter blocks loaded into the instrument. These classes are produced by a code-generator. This section describes the template for these classes.

The details of the command packet and parameter block formats are shown in the “ACIS Instrument Program and Command List,” MIT 36-01410 (IP&CL).

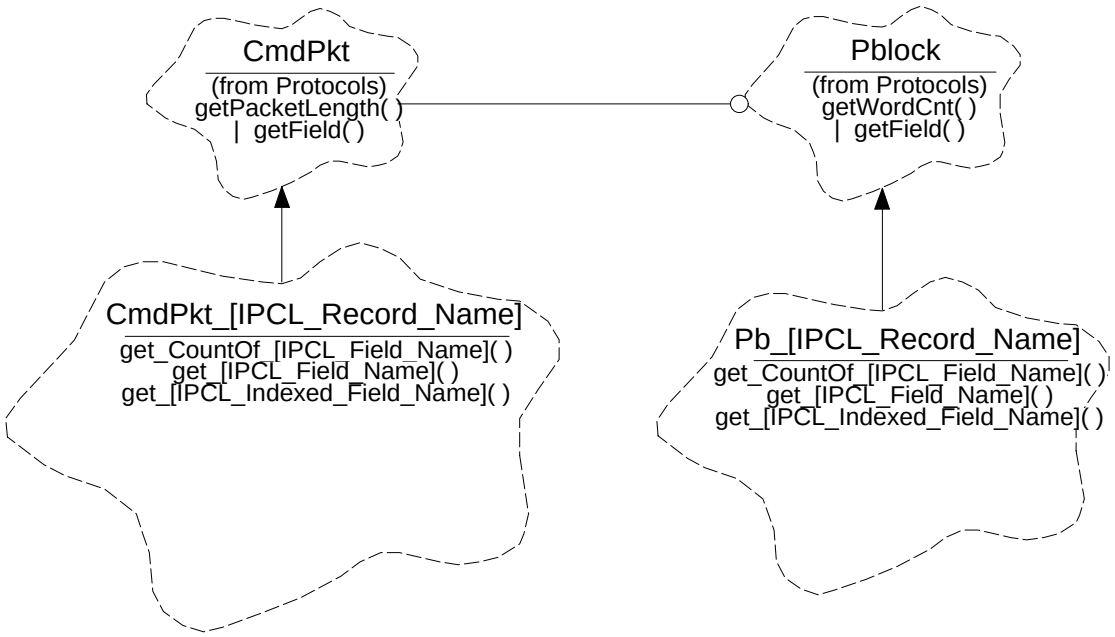
22.2 Uses

Use 1:: Read a particular field from a command or parameter block buffer
Use 2:: Determine the number of elements within a variable length command or parameter block

22.3 Organization

All generated command readers are a subclass of ***Protocols***::**CmdPkt** (see Section TBD). All generated parameter block readers are a subclass of ***Protocols***::**Pblock** (see Section TBD). The name of a given command packet reader class is the name of the record type, as listed in the IP&CL, with spaces replaced with underscores, and prepended with the name **CmdPkt_**. The name of a parameter block reader is the same, except it is prepended with the name **Pb_**.

FIGURE 98. Command and Parameter Block Reader Relationships



Protocols::**CmdPkt** - This class is a member of the ***Protocols*** class category and represents a generic command packet. It provides functions used by its subclasses to determine the total number of 16-bit words in the packet (`getPacketLength`) and to read a bit-field from within the packet's buffer (`getField`). This class is described in more detail in Section TBD.

Protocols::**Pblock** - This class is a member of the ***Protocols*** class category and represents a generic parameter block. It provides functions used by its subclasses to determine the total number of 16-bit words within the block (`getWordCnt`) and to read a bit-field from within the blocks buffer (`getField`). This class is described in more detail in Section TBD.

CmdPkt_[IPCL_Record_Name] - This represents a generic template for the various types of command packet classes, where "**[IPCL_Record_Name]**" is the name of the command packet record definition within the IP&CL database (spaces within the record name are replaced with "_" in the class name). Each class definition provides one member function for each field defined within the record (`get_[IPCL_Field_Name]`, `get_[IPCL_Indexed_Field_Name]`). For fields which are part of an array, this member function takes a single index argument, indicating which element of the array to read. For command packet records which end with a variable length array, the class provides a function which returns the number of elements within the array (`get_CountOf_[IPCL_Field_Name]`).

Pb_[IPCL_Record_Name] - This represents a generic template for the various types of parameter block classes, where "**[IPCL_Record_Name]**" is the name of the parameter block record definition within the IP&CL database (spaces within the record name are replaced with "_" in the class name). Each class definition provides one member function for each field defined within the record (`get_[IPCL_Field_Name]`, `get_[IPCL_Indexed_Field_Name]`). For fields which are part of an array, this member function takes a single index argument, indicating which element of the array to read. For parameter blocks which end with a variable length array, the class provides a function which returns the number of elements within the array (`get_CountOf_[IPCL_Field_Name]`).

22.4 Reader Design Issues

22.4.1 Assumptions

This section lists some of the assumptions made by the command packet and parameter block reader classes.

- No readable field is longer than 32-bits
- No command or parameter block contains more than 1 variable length array
- Variable length arrays are always at the end of a command or parameter block

22.4.2 Field Access Functions

Two approaches were considered when building the code-generator. One is to have the code-generator produce already customized code, which optimally read fields from the packet's or block's input buffer. The other approach has the code-generator producing standard code which relies on the compiler to optimize of the inline expansions of `getField`() member function calls. The current design takes the second approach.

The general form for every `get_[IPCL_Field_Name]` function consists of the following:

```
unsigned out;
out = getField(BITOFFSET, BITWIDTH, BITMASK, index, ARRAYWIDTH);

return out;
OR
return int(out << (16 - BITWIDTH- 1)) >> (16 - BITWIDTH - 1));
```

Where BITOFFSET is the bit position of the field from the start of the packet or block. If the field is within an array, BITOFFSET is the offset to the field within the first array element. BITWIDTH is the number of bits in the field and must be less than or equal to 32. BITMASK is a right-justified mask of the field (1's correspond to bits in the field). If the field is within an array, *`index`* is an argument which selects which array element to access and ARRAYWIDTH is the number of bits within one element of the array. If the field is not within an array, *`index`* and ARRAYWIDTH will be zero set to 0 by the code-generator.

The code-generator produces constants for all parameters to `getField`() except *`index`*. Given an appropriately written `getField`() member function, the compiler's optimizer can very efficient code when it is expanded.

22.4.3 Array Word Count Functions

Functions which return the number of elements in a variable length array at the end of a record consists of a converted version of the dimension formula of the field defining the array within the IP&CL. For Command Packets, these formula ususlly consist of the following:

```
((CmdPkt::getPacketLength() * 16) - ARRAYOFFSET) / ARRAYWIDTH
```

where ARRAYOFFSET is the bit-offset of the first element of the array from the start of the command packet, and ARRAYWIDTH is the number of bits within each element of the array. The factor of 16 converts the number of words in the packet to the number of bits.

The formula for parameter blocks is similar, except that they call **Pblock**::getWordCnt() instead of **CmdPkt**::getPacketLength().

22.5 Class CmdPkt_[IPCL_Record_Name]

Documentation:

This is is a template for all Command Packet reader classes generated from data contained within the ACIS Instrument Program and Command List database. The generated classes are responsible for providing accessor member functions for each field defined by the command format.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`CmdPkt`**

Public Interface:

Operations:
```
get_CountOf_[IPCL_Field_Name]()
get_[IPCL_Field_Name]()
get_[IPCL_Indexed_Field_Name]()
```

Concurrency: Guarded

Persistence: Transient

22.5.1 get_CountOf_[IPCL_Field_Name]()

Public member of: **CmdPkt_[IPCL_Record_Name]**

Return Class: **unsigned**

Documentation:

On commands which end with a variable length array of structures, this function is provided to compute and return the number of elements within the array.

Semantics:

In the general case, use the inherited `getPacketLength`() to get the total number of words in the command packet:

```
# elements = (length * 16) - arrayoffset DIV arraywidth
```

Where *`arrayoffset`* is the starting bit-position of the array, and *`arraywidth`* is the number of bits within each element of the array.

Concurrency: Guarded

22.5.2 get_[IPCL_Field_Name]()

Public member of: **CmdPkt_[IPCL_Record_Name]**

Return Class: **unsigned or int**

Documentation:

Each of these member functions return the contents of the corresponding field within the command packet. These types of fields are at a fixed position within the command packet.

Semantics:

Call the parent function **CmdPkt**::`getField`(), passing the constant *`bitoffset`*, *`bitwidth`*, *`bitmask`* as arguments. Pass 0 for the *`index`*, and 0 for the *`arraywidth`*. If the field is **unsigned**, return the read value. If the field is **signed**, sign-extend the result by shifting the word left to place the msb of the field in the msb of the return word, cast the value to an **int**, and right shift the word back to its original position.

Concurrency: Guarded

22.5.3 get_[IPCL_Indexed_Field_Name]()

Public member of: **CmdPkt_[IPCL_Record_Name]**

Return Class: **unsigned or int**

Arguments:

unsigned *index*

Documentation:

These functions return the value of the named field from within an array of structures. *index* indicates which structure to select.

Semantics:

Call the parent function **CmdPkt**::getField(), passing the constant *bitoffset*, *bitwidth*, *bitmask* as arguments. Pass the *index* argument, and the constant *bitsize* of the structure being indexed. If the field is **unsigned**, return the read value. If the field is **signed**, sign-extend the result by shifting the word left to place the msb of the field in the msb of the return word, cast the value to an **int**, and right shift the word back to its original position.

Concurrency: Guarded

22.6 Class Pb_[IPCL_Record_Name]

Documentation:

This is is a template for all Parameter Block reader classes generated from data contained within the ACIS Instrument Program and Command List database. The generated classes are responsible for providing accessor member functions for each field defined by the parameter block format.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`Pblock`**

Public Interface:

Operations:
```
get_CountOf_[IPCL_Field_Name]()
get_[IPCL_Field_Name]()
get_[IPCL_Indexed_Field_Name]()
```

Concurrency: Guarded

Persistence: Transient

22.6.1 get_CountOf_[IPCL_Field_Name]()

Public member of: **Pb_[IPCL_Record_Name]**

Return Class: **unsigned**

Documentation:

On parameter blocks which contain a variable length array of structures, this function is provided to compute and return the number of elements within the array.

Semantics:

In the general case, use the inherited `getWordCnt`() to get the total number of words in the parameter block.

```
# elements = (length * 16) - arrayoffset DIV arraywidth
```

Where *arrayoffset* is the starting bit-position of the array, and *arraywidth* is the number of bits within each element of the array.

Concurrency: Guarded

22.6.2 get_[IPCL_Field_Name]()

Public member of: **Pb_[IPCL_Record_Name]**

Return Class: **unsigned or int**

Documentation:

Each of these member functions return the contents of the corresponding field within the parameter block. These types of fields are at a fixed position within the parameter block.

Semantics:

Call the parent function **Pblock**::`getField`(), passing the constant *bitoffset*, *bitwidth*, *bitmask* as arguments. Pass 0 for the *index*, and 0 for the *arraywidth*. If the field is **unsigned**, return the read value. If the field is **signed**, sign-extend the result by shifting the word left to place the msb of the field in the msb of the return word, cast the value to an **int**, and right shift the word back to its original position.

Concurrency: Guarded

22.6.3 get_[IPCL_Indexed_Field_Name]()

Public member of: **Pb_[IPCL_Record_Name]**

Return Class: **unsigned or int**

Arguments:

unsigned *index*

Documentation:

These functions returns the value of the named field from within an array of structures. *index* indicates which structure to select.

Semantics:

Call the parent function **Pblock**::getField(), passing the constant *bitoffset*, *bitwidth*, *bitmask* as arguments. Pass the *index* argument, and the constant *bitsize* of the structure being indexed. If the field is **unsigned**, return the read value. If the field is **signed**, sign-extend the result by shifting the word left to place the msb of the field in the msb of the return word, cast the value to an **int**, and right shift the word back to its original position.

Concurrency: Guarded

23.0 Telemetry Writer Templates (36-53232.03 01)

23.1 Purpose

The purpose of the collection of telemetry packet writer classes is to format bit-packed fields into telemetry packet buffers to be sent out of the instrument. These classes are produced by a code-generator. This section describes the template for these classes.

The details of the telemetry packet formats are shown in the “ACIS Instrument Program and Command List,” MIT 36-01410 (IP&CL).

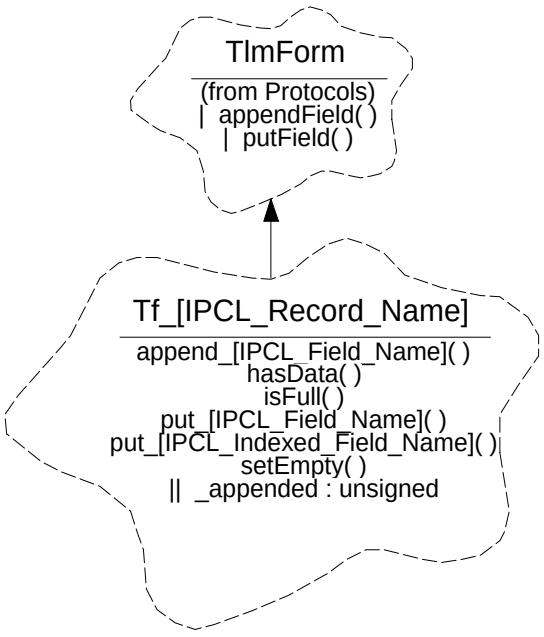
23.2 Uses

Use 1:: Write a particular field into a telemetry packet buffer
Use 2:: Append elements to the end of a telemetry packet buffer
Use 3:: Determine the number of words in a telemetry packet buffer
Use 4:: Determine if a variable length telemetry packet buffer is full
Use 5:: Determine if a variable length telemetry packet buffer has any data to send

23.3 Organization

All generated telemetry writers are a subclass of ***`Protocols`***::**`TlmForm`** (see Section TBD). The name of a given telemetry packet writer class is the name of the record type, as listed in the IP&CL, with spaces replaced with underscores, and prepended with the name **`Tf_`**.

FIGURE 99. Telemetry Packet Writer Relationships



Protocols::**TlmForm** - This class is a member of the ***Protocols*** class category and represents a generic telemetry packet formatter. It provides functions used by its sub-classes to determine the xxx. This class is described in more detail in Section TBD.

Tf_[IPCL_Record_Name] - This represents a generic template for the various types of telemetry packet formatter classes, where "**[IPCL_Record_Name]**" is the name of the telemetry packet record definition within the IP&CL database (spaces within the record name are replaced with "_" in the class name). Each class definition provides a member function which returns the number of words in the packet (`getWordCount`) and one member function for each field defined within the record but are not part of a variable length array (`put_[IPCL_Field_Name]`, `put_[IPCL_Indexed_Field_Name]`). For fields which are part of a fixed length array, this member function takes a single index argument, indicating which element of the array to read. If a field defines a variable length array at the end of a packet, the class contains a member function which appends all elements of the field to the end of the packet (`append_[IPCL_Field_Name]`). If the field's data type is another record, the function arguments consist of the fields within the defined record. If a telemetry packet is variable in length, the class contains member functions which indicate if the packet has had any data appended to it (`hasData`), and whether or not the packet has room for any more data (`isFull`).

23.4 Writer Design Issues

23.4.1 Assumptions

This section lists some of the assumptions made by the command packet and parameter block reader classes.

- Except for fields defining structures, no single field is longer than 32-bits
- No telemetry packet contains more than 1 variable length array
- Variable length arrays are always at the end of a telemetry packet buffer

23.4.2 Put Field Access Functions

Two approaches were considered when building the code-generator. One is to have the code-generator produce already customized code, which optimally writes fields to the packet's output buffer. The other approach has the code-generator producing standard code which relies on the compiler to optimize of the inline expansions of `putField`() member function calls. The current design takes the second approach.

The general form for every `put_[IPCL_Field_Name]` function consists of the following:

```
putField(value, BITOFFSET, BITWIDTH, BITMASK, index, ARRAYWIDTH);
return;
```

Where *value* is the value to store into the telemetry buffer, BITOFFSET is the bit position of the field from the start of the packet or block. If the field is within an array, BITOFFSET is the offset to the field within the first array element. BITWIDTH is the number of bits in the field and must be less than or equal to 32. BITMASK is a right-justified mask of the field (1's correspond to bits in the field). If the field is within an array, *`index`* is an argument which selects which array element to access and ARRAYWIDTH is the number of bits within one element of the array. If the field is not within an array, *`index`* and ARRAYWIDTH will be zero set to 0 by the code-generator.

The code-generator produces constants for all parameters to `putField`() except *`value`* and *`index`*. Given an appropriately written `putField`() member function, the compiler's optimizer can very efficient code when it is expanded.

23.4.3 Append Field Access Functions

The approach used to append items to the end of a telemetry packet is similar to that taken to write a value into the middle of the packet buffer, except that the append_[IPCL_Field_Name] functions may append entire records to the end of the buffer.

The general form for each append_[IPCL_Field_Name] function consists of one call to appendField() for each subfield in the structure being appended. The subfields are appended in the order they appear in the appended structure.

23.5 Class Tf_[IPCL_Record_Name]

Documentation:

This is a template for all telemetry packet writer classes generated from data contained with the ACIS Instrument Program and Command List database. The generated classes are responsible for providing functions which write or append fields to a telemetry packet buffer.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`TlmForm`**

Public Interface:

Operations:
```
append_[IPCL_Field_Name]()
hasData()
isFull()
put_[IPCL_Field_Name]()
put_[IPCL_Indexed_Field_Name]()
setEmpty()
```

Private Interface:

Has-A Relationships:

`unsigned` *_appended*: This instance variable is defined in all variable length and indicates the number of elements which have been appended to the end of the packet buffer.

Concurrency: Guarded

Persistence: Transient

23.5.1 append_[IPCL_Field_Name]()

Public member of: **`Tf_[IPCL_Record_Name]`**

Return Class: **`void`**

Arguments:

```
unsigned or int subfield1
unsigned or int subfield2
...
```

Documentation:

This function appends the elements of the indicated field to the end of the telemetry packet buffer. Each passed argument corrresponds to each sub-field of the appended structure.

Semantics:

For each passed argument, call `appendField`(), passing the value to store, the bit-offset of the field in the first array element, the bit-width of the field, the index of the structure being appended to and the bit-width of a single structure element. Once the element has been appended, increment *_appended*.

Concurrency: Guarded

23.5.2 hasData()

Public member of: **Tf_[IPCL_Record_Name]**

Return Class: **Boolean**

Documentation:

This function is provided by all variable length telemetry packets and indicates whether or not data has been appended to the packet. If the packet has had data appended, the function returns *BoolTrue*. If not, it returns *BoolFalse*.

Semantics:

Return *BoolTrue* if *_appended* is not 0, otherwise return *BoolFalse*.

Concurrency: Guarded

23.5.3 isFull()

Public member of: **Tf_[IPCL_Record_Name]**

Return Class: **Boolean**

Documentation:

This function is provided by all variable length telemetry packets and indicates whether the caller can append 1 more element to the buffer.

Semantics:

Compute the number of bits occupied if the buffer had 1 more element appended.

```
offset = ARRAYOFFSET + (_appended + 1) * ARRAYWIDTH
```

If it is beyond the capacity of the buffer, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

23.5.4 put_[IPCL_Field_Name]()

Public member of: **Tf_[IPCL_Record_Name]**

Return Class: **void**

Arguments:

unsigned or int *value*

Documentation:

This function writes the passed *value* into the telemetry packet buffer to the indicated bit-field. If the field is **unsigned**, the value argument is an unsigned number. If the field is **signed**, then value is passed as a signed integer.

Semantics:

Call the parent function **TlmForm**::putField(), passing the *value*, the constant *bitoffset*, *bitwidth*, and *bitmask* as arguments, Pass 0 for the *index*, and 0 for the *arraywidth*.

Concurrency: Guarded

23.5.5 put_[IPCL_Indexed_Field_Name]()

Public member of: **`Tf_[IPCL_Record_Name]`**

Return Class: **`void`**

Arguments:

`unsigned or int` *`value`*
`unsigned` *`index`*

Documentation:

This function writes the passed *`value`* into the telemetry packet buffer to the indicated array field. If the field is **`unsigned`**, the *`value`* argument is an unsigned number. If the field is **`signed`**, then value is passed as an signed integer. *`Index`* indicates into which array element to store the value.

Semantics:

Call the parent function **`TlmForm`**::`putField`(), passing the *`value`*, the constant *`bitoffset`*, *`bitwidth`*, and *`bitmask`* as arguments, Pass the provided *`index`* argument, and the constant for the *`arraywidth`*.

Concurrency: Guarded

23.5.6 setEmpty()

Public member of: **`Tf_[IPCL_Record_Name]`**

Return Class: **`void`**

Documentation:

This function is placed in all variable length classes. When called, the function resets the count of number of elements current appended to the telemetry packet buffer.

Semantics:

_appended = 0

Concurrency: Guarded

24.0 Huffman Table Data Compression (36-53233 A)

24.1 Purpose

The Huffman data compression function provides the capability to significantly reduce the telemetry required to transfer the acquired data.

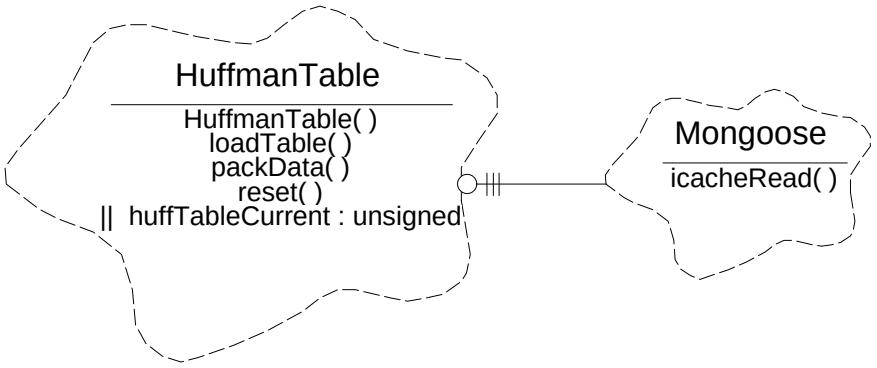
24.2 Uses

Use 1:: A method of copying the selected Huffman table from I-Cache to D-Cache.
Use 2:: A method of converting a set of values to packed Huffman codes.
Use 3:: A method of packing uncompressed data values.
Use 4:: A method of processing with truncated Huffman codes.
Use 5:: A means of appending additional buffer of codes to an existing one.

24.3 Organization

Figure 1 illustrates the relationship between the classes used by the Huffman Table Data Compression.

FIGURE 100. Huffman Table Data Compression Class Relationships



The HuffmanTable uses the ***Executive*** and ***Protocols***, class categories.

HuffmanTable- This function is responsible for converting data into Huffman codes.

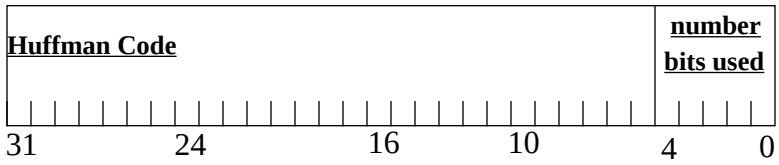
Mongoose - This class provides a quick copy from I-Cache to another memory location. It is a subclass of ***Executive***.

24.4 The Huffman Table

A Huffman table is generated by determining the statistical frequency of occurrence of each of the values which makeup the data set. The codes are distributed according to each values position in that distribution. The values which appear most frequently are given the shortest Hoffman code strings (number of bits), those least frequently encountered are assigned the longer codes. Each table entry consists of a set of bits (code) and the number of bits in that set. To use the table; as each value in the data being compressed is encountered, its corresponding code is installed (abutted) in the compressed data buffer. The number of words compressed and a buffer filled with adjacent variable length sets of ones and zeros results.

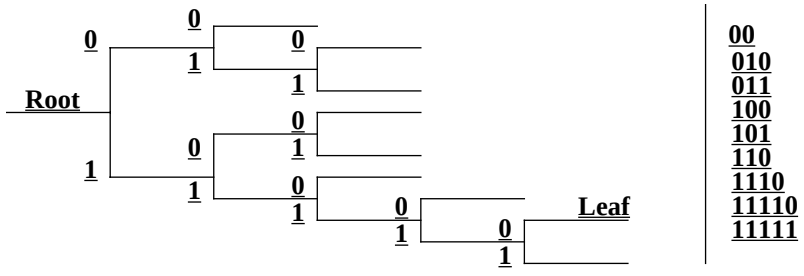
The Huffman table to be flown[1], consists of an array of words in which the low order five bits provide the number of relevant bits (left adjusted, zero filled) in the high order bits of that word. Refer to Figure . The bits will be packed into the output word in little endian order. Since decoding is anticipated using a tree structure, the bits must be ordered from the root, through the branches, (nodes) to a leaf. The leaf will provide the original value.

FIGURE 101. Data Distribution in Huffman Table Word



As stated, the order of words in the array is based on the statistical probability of that data value occurring in the data set. The most frequently occurring values have codes with the fewest bits. Decoding is accomplished using a logical tree structure corresponding to the compression table. Each bit extracted from the coded word determines which of the binary nodes (branch) of the tree is to be taken. When no additional node is available, (a leaf is located) that code has been determined. The data value at that location is the uncompressed value. Refer to Figure 102.

FIGURE 102. Sample Decoding Tree



1. The flight table is noted in: *CCD Bias Level Determination Algorithms*, 36-56101 Version 02, Section 5.0

24.4.1 Optimizing the Flight Table

The table is optimally tuned to the statistical probability of the frequency of occurrence of data values across the range of values anticipated. While the overall variation in some of the data may be large, the variation between adjacent pixel values is expected to be small. To further enhance the probability that most values would correspond to the shortest codes, the difference between adjacent good pixels values was used in creating the table. An initial nominal value may be provided to obtain a difference with the first pixel value. The average over-clock value is a good candidate. Bad pixels and those with parity errors are provided separate values. With twelve bits of data there are 4095 values. The possible differences are plus and minus 4095 or 8190 values; except that 4095 and 4094 are pre-empted for bad pixel and bad bias respectively, so there is no corresponding -4095 or -4094, and there is no -0. The table length is 8187 words. The bad bias and bad pixel Huffman codes are stored in the header. The values are offset to eliminate negative table indices.

To reduce the size of the table, a cutoff in the data distribution may be determined and a huffman code assigned to identify the occurrence of an accompanying literal value.

Each table is composed of a header followed by the table, itself. The header defines a unique identifier, the index of the first huffman code, the table size, and the Literal, Bad-Bias and BadPixel codes.

24.4.1.1 Characteristics of a Truncated Huffman Table

A Literal Value is any value who’s Huffman index is not available. While calculated in the usual manner; differenced from its neighbor and offset to provide only positive values, it falls in the region which has been truncated. To transmit this datum, a Literal Code will be appended to the differenced, offset, Literal Value and the combination will be packed for transmission in the usual manner. Because this outlying value could result in two combination codes being transmitted, nominal to outlier - outlier to nominal, the outlier value is not retained for differencing with the next pixel value. That next value will be differenced with the prior nominal value, the one used in determining the outlier Huffman index.

In a truncated Huffman table, as in a full table, the flagged conditions, BAD_BIAS and BAD_PIXEL, will occupy the fifth and sixth header entries. The forth entry is the truncated table Literal Index. The actual pixel value will be appended to this code before being packed. Each truncated table entry is for indices which represent themselves, indexing the

lowest through the highest indices for which a “real” Huffman compression code is available. These entries have the same form as full table entries.

When creating a pseudo Literal Code/Value Huffman compression word, the Literal Code shall not exceed fifteen bits in length. The selection of the Literal Code should reflect the probability of the frequency of occurrence of the total number of truncated values.

NOTE: Because the last word in a packed buffer may not be filled, and because zeros are relevant values, the data structure headers of ALL packets which may contain Huffman packed data must contain a count of the number of items packed, or the number of items must be known a priori.

24.5 Scenarios

24.5.1 Operational Overview

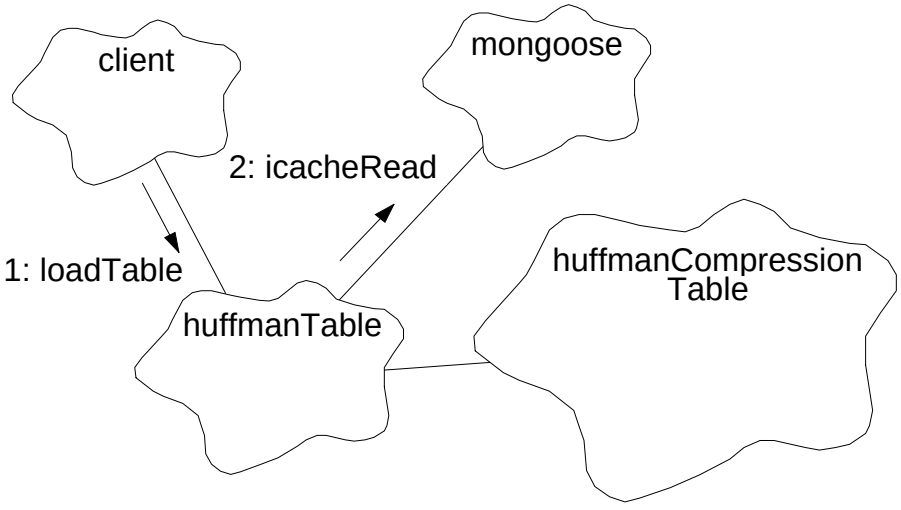
The client will call the constructor to initialize values. Before attempting to compress data, the designated Huffman Table will be selected and be copied from I-Cache into the predetermined location in D-Cache. Only one table is available at one time. Therefore, the statistical distribution of each CCDs' differences must be similar or this restriction may not provide the most efficient compression. This decision was based on the large table size.

When the Huffman data compression process is called by a client, the client provides a list of data values to be concentrated and a buffer of certain length in which to load the compressed data. The process will compress the values from the list until either; the list is exhausted, or the output buffer is filled. Each compressed item is packed (abutted) into the output buffer word. As each word is completed, any overflow will be used to start the next word. Upon return, the process will provide the number of items compressed and the number of output words filled. It is the clients prerogative to initiate a new buffer for the concentration of additional data or to append the additional data into the existing buffer. To do the former, it will re-initialize key variables, provide the additional data values, and a new buffer address. By delivering more data to the process while continuing the pointer to contiguous buffer space, the client will satisfy conditions for appending to a buffer. Appending across different buffers is possible. It is not intended. Any corruption of the compressed data buffer/s renders accurate decompression of the entire buffer/s difficult if not impossible, therefor a check-sum of some type is advisable.

24.5.2 Use 1: Loading a Huffman Table from I-Cache into D-Cache

Several Huffman tables will be located in I-Cache. A client science process e.g. `biasThief`(), will select the most appropriate Huffman compression table to be referenced by all concurrent data compressing tasks.

FIGURE 103. Load Huffman Table to D-Cache

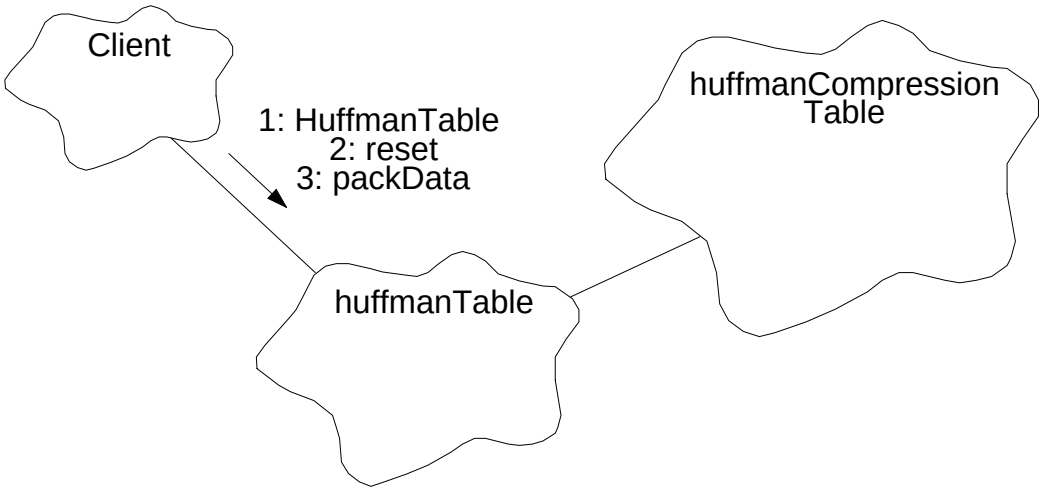


The process will use 1: the ***HuffmanTable*** function `loadTable` to have that table copied into D-Cache. `loadTable` will pass that address, and HUFF_TABLE_ADDR, the constant D-Cache location for the table, to the 2: **Mongoose** quick copy function `icacheRead` which will read the table array in I-Cache and write it into D-Cache. refer to Figure 103, above.

24.5.3 Use 2: Converting a Set of Values to Packed Huffman Codes.

The Science client will derive or obtain the data to be compressed. The data is presumed by **HuffmanTable** to be a set of pixel or bias map values stored in the low 12 bits of the input words[1]. The client will also supply a buffer of the type corresponding to the clients purpose. As shown in Figure 104, the client will 1: call the **HuffmanTable** constructor to obtain an instance of the function which will initialize some variables. The 2: `reset` function should be used whenever loading a fresh buffer. It re-initializes values retained from prior packing which are used when appending. The function will then 3: use the **HuffmanTable** function `packData` passing a reference to the data values to be converted and a reference to the buffer available to be filled along with the length of each. The proper Huffman code array is presumed to have been loaded into the predetermined HUFF_TABLE_ADDR location in D-Cache.

FIGURE 104. Convert Data to Packed Huffman Codes Using The Huffman Compression Table



The function will extract the low 12 bits from input words as the value to be compressed. It will test for the special bad bias and bad pixel values, then deliver that values array word if either is present. Otherwise, it will deliver the array word corresponding to the difference between the prior value (initially zero) and the current value, adjusted with an offset to account for possible negative values. The Huffman code bit count is extracted (removed) from the word, and the code is installed in the unused portion of the packed word being built. Codes are abutted until the word is filled or overflowed. After obtaining

1. For further information on Pixel and Bias hardware processing, refer to: *DPA Hardware Specification & System Description*, 36-02104 Rev B section 2.2.2.5 through 2.2.2.5.6.

the next output buffer word, the function installs the overflow bits beginning the next compressed word (refer to Figure). It will pack additional codes until this word has been filled. As each value is converted and installed, or each buffer word is filled, the count of items or buffer words is decremented. This process continues until the set of items to convert and pack is exhausted or until the available buffer is filled. If the last code to be loaded overflows the last available buffer word, the count of items compressed is decremented, and the remaining bits, their number, and the previous word value are retained in anticipation of appending overflowed bits into the next buffer provided. If the input data has been exhausted and the available space in the last output word not used, the function returns BoolTrue, indicating that additional space is available in that word. If the last code overflowed the available space, its pixel value is not counted and the space is deemed available (unused). If the word is filled exactly, the function returns BoolFalse.

With the data packed or the buffer filled, the function will return the state of the last word packed, and it will make available the number of items remaining to be packed (if any) and the number of words remaining unused in the buffer. To begin another buffer, the client uses **`HuffmanTable`**::`reset` before delivering a fresh list of values to be compressed and packed.

The function **`HuffmanTable`**::`getTableId` will return its Identifier from the header.

24.5.4 Use 3: A method of packing uncompressed data values

The same resource used to pack compressed data is used to pack uncompressed 12 bit data. After the 12 bit value has been extracted, correlation with a compression string is skipped and the full value is packed. Data is packed uncompressed after the compression table address has been specified as NULL. A table may be loaded, but it is disregarded.

24.5.5 Use 4: A method of processing with truncated Huffman codes.

Having provided the Huffman compressor with the proper arguments to acquire a truncated Huffman table; I-Cache address, length, and lowest index value, the packing of pixel values using real Huffman codes or constructed Literal Code/Value combinations will be invisible to the client. The decompression processor, when encountering the Literal Code, must extract and pass the Literal Value rather than passing the index normally associated with that code. Refer to Section 24.4.1.1 for additional details concerning the table itself.

24.5.6 Use 5: Appending Additional Codes to a Buffer

Without reinitializing with `reset`, `packData` is predisposed to continue as though it is appending to the last buffer returned to the client. If it has used the last output buffer word, the last value code attempted was partially loaded in the final word of that buffer but was not counted. The overflowed code segment is preserved across calls, and would be inserted as the first bits packed in the new buffer. The item would be counted and the first item value, corresponding to the code segment just installed, would be ignored. The function presumes that it is loading the next word of a sequential buffer. Additional compressed words would be abutted to this segment.

If input data remained and the last output word was used, yet the return state indicates space in the last word, the word has overflowed. To append, the client should call designating the next word as the beginning of the output buffer. The new input should begin with the (uncounted) last pixel value. The input data may be augmented.

If all the input data was compressed, without overflowing the output buffer, and an append request was delivered, the function would begin by obtaining the first values' code. It would then pack the code into the first output buffer word at the location just past that in which the last word was packed on the assumption that the first word of the specified buffer was the last word it had processed previously.

If all the input data was used and the state returned indicated that the last output word used was exactly filled, the client should designate the next word as the beginning of the output buffer when appending. If the state showed additional space, the client should return the last word used as the new first word of the output buffer to continuing appending.

24.6 Class HuffmanTable

This Class provides Huffman compression for data transmission.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:
`HuffmanTable()`
`getTableId()`
`loadTable()`
`packData()`
`reset()`

Private Interface:

Has-A Relationships:

unsigned *`accOut`*: this variable is the accumulator for concatenated Huffman codes. When filled it is copied to the output buffer. When the buffer fills, this variable holds the bits which overflowed the last word in the output buffer. These bits are retained to facilitate appending codes. When the client appends data to a buffer, This value is inserted into the first word.

unsigned *`highLimit`*: This variable, derived from *lowLimit* and *tableSize* (from the header), is the high index of the truncated Huffman table used for compression. Processed pixel values; raw events which have been differenced and offset, and which are greater than this value will be appended with the Huffman Literal Code before being packed.

unsigned *`huffTableCurrent[]`*: This is an array containing the currently used Huffman look-up table. To reduce the amount of memory consumed in D-Cache by the tables, this buffer is shared by all **`HuffmanTable`** instances. Therefore, all active compression instances must be using the same lookup values. The designated table, which is stored in I-Cache before launch, is loaded via the static member function, `loadTable`().

unsigned *huffTableCurrentRef*: This variable contains a copy of the I_Cache pointer referencing the address of the Huffman Table desired for this instance. A value of zero indicates that uncompressed pixel data is to be packed.

static const unsigned * *huffTablePtr*: This pointer contains the I-Cache address of the table which was copied to D-Cache regardless of which instance copied it.

unsigned *huffTruncCode*: In Truncation Mode, This variable contains the converted contents of the Huffman Literal Index in which the bit count has been increased by 12. This code is used to encapsulate processed pixel values which do not have explicit indices of their own in a truncated Huffman table The original value was extracted from the header. Note: the original Huffman code provided as host for the Literal value, may not contain more than 15 Huffman code bits.

unsigned *leftOut*: This variable holds the number of bits which overflowed the last word in the output buffer. It is retained to facilitate appending codes.

unsigned *lowLimit*: This variable, located in the header of a Huffman table, contains the low index of the Huffman table used for compression. Processed pixel values; raw events which have been differenced and offset, which are less than this value will be prepended with the Huffman Literal Code before being packed. In a full table this value is zero.

unsigned *numLast*: This contains the previous non-special pixel value used to obtain the difference between it and the current value. It is retained to facilitate appending codes. This value is initialized by `reset`.

unsigned *numOut*: This variable contains the running total of bits packed in the current compressed (output) word when the last data (input) word was processed. It is the number of bits already loaded in the first buffer word when appending with new data words. It is used to facilitate appending codes.

Concurrency: Guarded

Persistence: Transient

24.6.1 HuffmanTable()

Public member of: **`HuffmanTable`**

Documentation:

This constructor function uses `reset` to initialize most of the instance variables. *huffTableCurrentRef* is initialized to zero, no compression, just pack the data. The unestablished table parameters, *lowLimit*, *highLimit* and *huffTruncCode* are set to zero.

Concurrency: Sequential

24.6.2 getTableId()

Public member of: **`HuffmanTable`**

Return Class: **`unsigned`**

Documentation:

This function returns the unique table identifier contained in the header.

Concurrency: Guarded

24.6.3 loadTable()

Public member of: **`HuffmanTable`**

Return Class: **`void`**

Arguments:

`const unsigned *` *`icacheAddr`*

Documentation:

This function is used to load the designated Huffman table located at *icacheAddr*, from I_Cache into the shared lookup table in D-Cache as *`huffTableCurrent`*, employing the **Mongoose** quick copy routine, `icacheRead`. If the pointer *`huffTablePtr`* matches *`icacheAddr`*, the desired table is currently loaded, so it is not reloaded. If *`icacheAddr`* is zero, that value is installed in *`huffTableCurrentRef`* and the table in D-Cache is not modified.

When a Huffman table is to be loaded, the limiting indices are established from parameters in the header. *lowLimits specified. highLimit* is computed as *lowLimit* plus *tableSize* minus 1. The Huffman code in the Literal index is restructured as host, *huffTruncCode*, for any 12 bit differenced and offset pixel value with an index which falls in the truncated region.

Preconditions:

The tables must begin on word boundaries.

Concurrency: Guarded

24.6.4 packData()

Public member of: **HuffmanTable**

Return Class: **Boolean**

Arguments:

```
const unsigned short * unpackPtr
unsigned & unpkLength
unsigned * compressPtr
unsigned & comprLength
```

Documentation:

This function will map each data word in the buffer pointed to by *unpackPtr*, with the corresponding Huffman code from *huffTableCurrent* and will pack that code into the output buffer pointed to by *compressPtr*. When packing the buffer, it will be able to initiate a new Huffman code series or append additional data as part of a continuing series. Designated input will be compressed into the output buffer as long as both input words and output space are available. This function considers only the least significant 12 bits of input words to be valid data. The input buffer length, *unpkLength*, and output buffer length, *comprLength*, are decremented as each type of word is processed. These arguments are available to the client upon return. The function returns the state of the last buffer word. The state is BoolTrue if the last word has available space, or BoolFalse if it was exactly filled. If not filled and it was the last output word, yet input data words remain, the last word attempted overflowed the output word. `packData` will pack 12 bit words without compression if the table pointer is NULL. When a truncated table is in use; this function will install any Literal value which has no target index, into the created Huffman code host prior to packing that code.

Preconditions:

The input, output, and table pointers must refer to an appropriate word boundary. The Huffman code length must be in the range 1 through 27 bits.

Postconditions:

The count of remaining input words and remaining output words is available to the client. The state of the last output word is returned as BoolTrue if more codes can be appended to it, else as BoolFalse if full.

Concurrency: Guarded

24.6.5 reset()

Public member of: **`HuffmanTable`**

Return Class: **`void`**

Documentation:

This function initializes instance variables, *`numLast`*, *`numOut`*, *leftOut*, and *`accOut`* when the client desires to begin loading a buffer rather than appending more data to an existing buffer.

Concurrency: Guarded

25.0 Front End Processor Management Classes (36-53217 B)

25.1 Purpose

The purpose of the Front End Processor Management classes, which run on the Back End Processor, are to manage the shared-memory protocol between the Back End Processor and each of the Front End Processors, and to provide general purpose Front End Processor science and diagnostic functions.

This section describes two classes, the **`FepManager`** class, and the **`FepIoManager`** class.

25.2 Uses

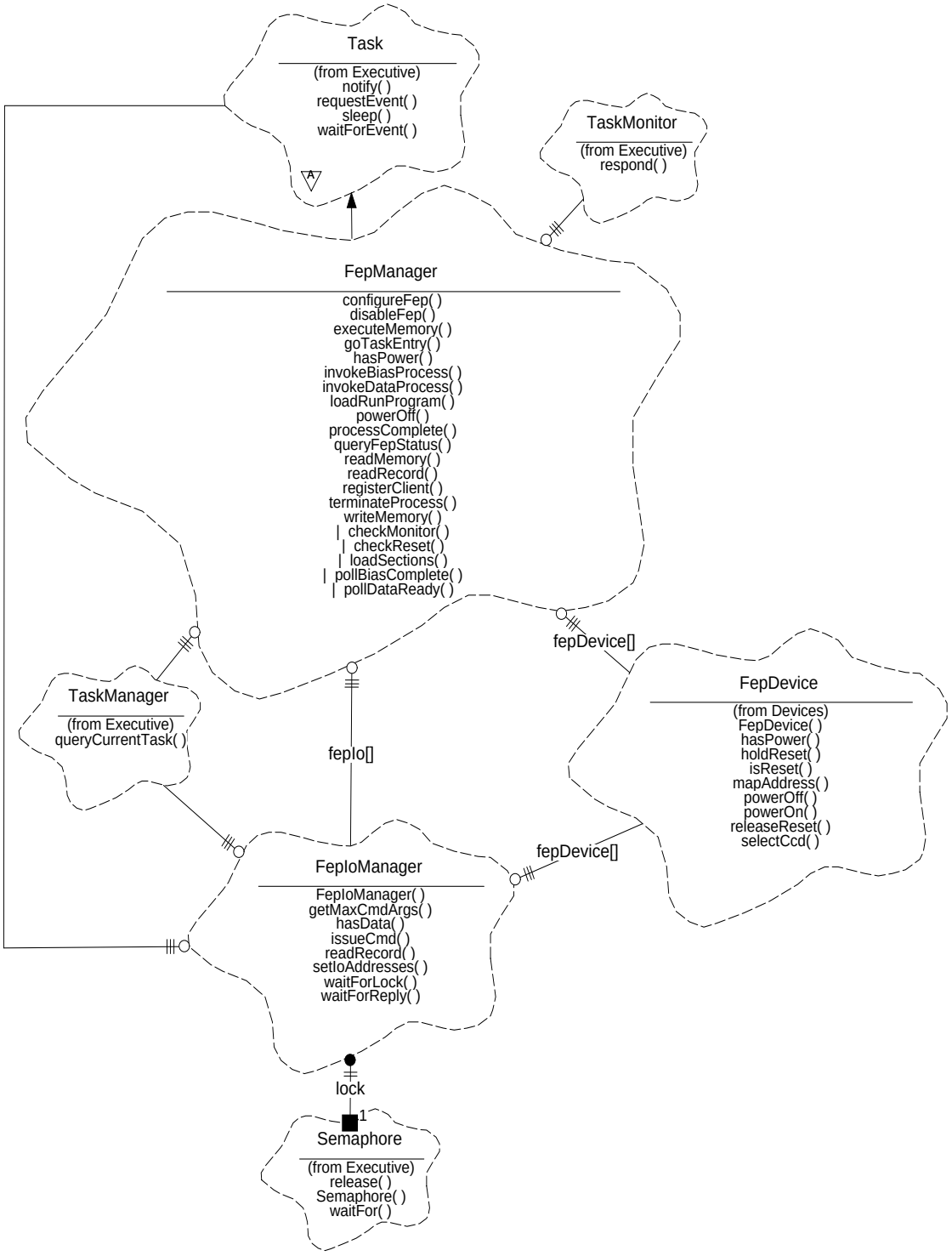
The following lists the primary uses of the Front End Processor Management classes:

Use 1:: Control Power to each of the Front End Processors
Use 2:: Reset, load, and run a program on a Front End Processor
Use 3:: Read a Front End Processor's Memory
Use 4:: Write to a Front End Processor's Memory
Use 5:: Execute a subroutine loaded into a Front End Processor
Use 6:: Configure a Front End Processor for a science run
Use 7:: Start bias calibrations on all configured Front End Processors
Use 8:: Start data processing on all configured Front End Processors and acquire science data from the Front End Processors
Use 9:: Stop bias or data processing on all configured Front End Processors
Use 10:: Disable a Front End Processor for use by science processing
Use 11:: Write a bad pixel code and its parity into the pixel bias map and parity plane

25.3 Organization

Figure 105 illustrates the class relationships used by the Front End Processor Manager, **`FepManager`**, and the I/O Management class, **`FepIoManager`**.

FIGURE 105. FEP Manager and I/O Manager Classes



`FepManager` - This class is a subclass of ***`Executive`***::**`Task`** and is responsible for the overall management of the Front End Processors. This class provides functions to power off all Front End's and to run new programs (`powerOff`, `loadRunProgram`), to read and write FEP memory and to execute subroutines loaded within a FEP (`readMemory`,

writeMemory, executeMemory), to configure a FEP to prepare for a science run (configureFep), to start FEP bias and data processing operations (invokeBiasProcess, invokeDataProcess), and to acquire data from the running FEPs and to stop current FEP processes (consumeData, terminateProcess). There is only one instance of this class within ACIS, called *fepManager*.

FepIoManager - This class is responsible for providing access to the Front End Processor's shared-memory Command Mailbox and Science Data Ring Buffer. There is one instance of this class for each Front End Processor in the instrument. This class provides functions, primarily for use by the **FepManager**, to determine the maximum amount of data that can be sent to the FEP via its command mailbox (getMaxCmdArgs), to determine if the FEP's ring buffer contains any data (hasData), to issue a command to the FEP and wait for its response (issueCmd), to read data from the FEP's ring buffer (readData), and to establish the mailbox and ring buffer addresses used by the program currently running in the FEP (setIoAddresses). Access to instances of this class are performed via an array of pointers, *fepIo*[]. The array is indexed by the Front End Processor enumeration, **FepId**.

FepCallback - This class is a subclass of ***Devices***::**DevCallback**. It is used by the **FepManager** to obtain control during the processing of Front End Processor Interrupts. Its invoke() member function is responsible for passing control to the **FepManager**'s serviceDevice() function during Front End Processor Interrupt handling. There is one instance of this class within ACIS, called *fepCallback*.

DevCallback - This abstract class is defined by the ***Devices*** class category, and is responsible for defining the common interface to all device interrupt callback classes. It is described in more detail in Section 6.0.

FepDevice - This class is responsible for providing access to the control hardware for each Front End Processor. There is one instance of this class for each Front End Processor in the system. Both the **FepManager** and **FepIoManager** classes use this class to query and control the Front End Processor hardware registers. Access to instances of this class are performed via an array of pointers, *fepDevice*[]. The array is indexed by the Front End Processor enumeration, **FepId**. This class is defined in more detail in Section 10.0.

TaskManager - This class is defined by the ***Executive*** class category and is responsible for managing the overall task scheduling. It is used by the **FepManager** and **FepIoManager** classes to obtain access to the currently running task (queryCurrentTask()).

TaskMonitor - This class is defined by the ***Executive*** class category, and is a subclass of **Task**. It is responsible for polling each task in the instrument, and maintaining the watchdog timer. If any task fails to respond in a timely fashion, this task will fail to touch the watchdog, and the instrument will be reset. See Section 15.0 for more detail.

Task - This abstract class is defined by the ***Executive*** class category, and responsible for representing a thread of control within the instrument. The **FepManager** is a sub-

class, and also contains a pointer to an instance of this class (*clientTask*). See Section 15.0 for more detail.

Semaphore - This class is defined by the ***Executive*** class category. Each **FepIoManager** instance contains a **Semaphore** instance (*lock*) and uses it to arbitrate for access to the Front End Processor's Command Mailbox. See Section 15.0 for more detail.

25.4 Miscellaneous Items

25.4.1 FepManager Auxiliary Service Routine task

The **`FepManager`** uses a task to poll for requests and conditions (i.e. this assumes that reaction times are on the order of a half a second) from the Front End Processors. This task consists of a main polling loop, which sleeps for 0.2 second (TBD) on each iteration. From a black-box point of view, the existence of this task is transparent to the client code, and is used only as part of the manager's internal implementation.

25.4.2 FEP Mailbox and Ring-buffers

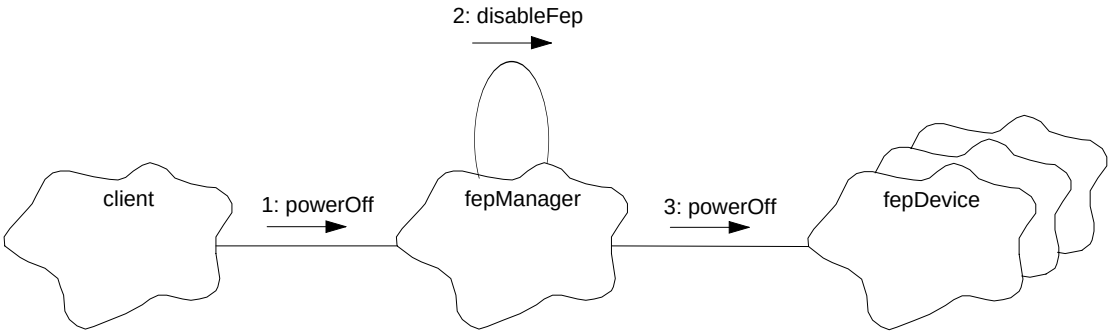
The software running on each Front Processor establishes a Command Mailbox and a science data ring-buffer in its shared memory space. The mailbox consists of a state variable, a length field, and data. The state variable controls what is currently contain in the mailbox, and which processor has write access to the box. The length field indicates how much data is in the mailbox. The science data ring buffer consists of a series of blocks, where each block consists of thirty two, 32-bit words. The ring buffer uses a shared-memory control structure containing a read index and write index. As the FEP writes blocks into the ring buffer, it advances the write index. As the BEP reads blocks, it advances the read index. When the read and write indices are the equal, the ring buffer is empty. When the write index is equal to the slot just prior to the read index, the ring buffer is full. For a more detailed description of the structure and content of BEP to FEP interface, refer to Section 4.9 and Section 4.10.

25.5 Scenarios

25.5.1 Use 1: Power off a FEP

Figure 106 illustrates the sequence of events when a client instructs the FEP Manager to turn off power to a Front End Processor. NOTE: Powering off a FEP destroys any work-in-progress and information contained on that FEP, including the pixel bias map values.

FIGURE 106. Power off FEP

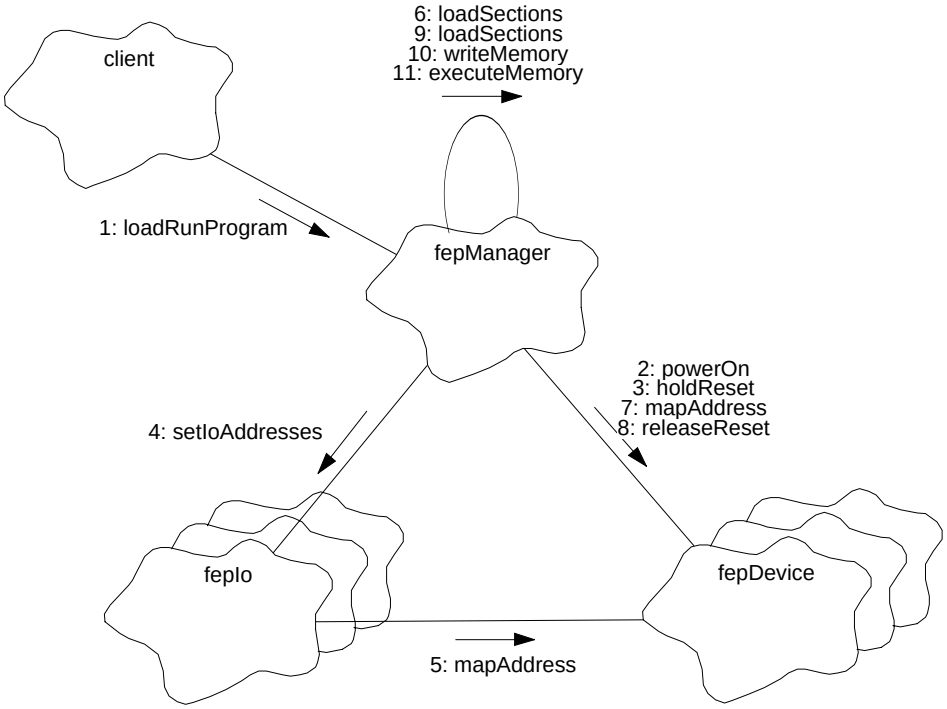


1. The *client* instructs the FEP Manager to turn off power to a Front End Processor, using *fepManager*.powerOff().
2. *fepManager*.powerOff() calls disableFep() to take the current FEP out of its list of enabled FEPs.
3. The loop body then indexes the FEP device pointer, and invokes its *fepDevice*.powerOff() function to turn off the power to the Front End Processor.

25.5.2 Use 2: Reset, load, and run a program on a Front End Processor

Figure 107 illustrates the actions used to reset a Front End Processor, and to load and run a program on a Front End Processor. In this diagram, *fepIo* represents instances of the **FepIoManager** class.

FIGURE 107. Resetting and loading and running a program on a FEP



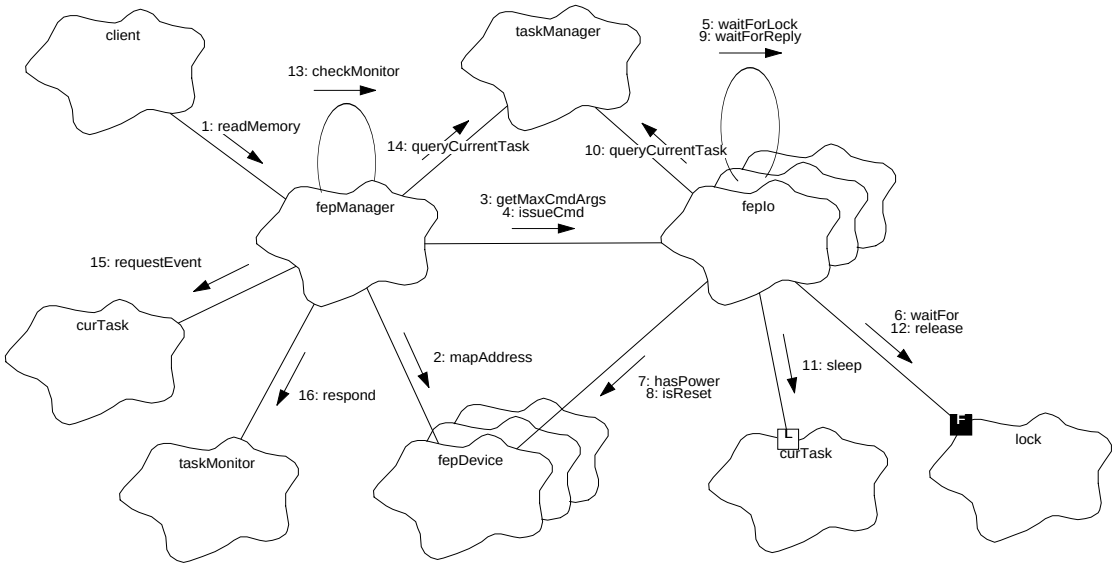
1. The client instructs the FEP Manager to reset the FEP, and load and run a program, using *fepManager*.loadRunProgram().
2. loadRunProgram() ensures that the FEP has power, using *fepDevice*.powerOn()
3. loadRunProgram() ensures the FEP is reset, using *fepDevice*.holdReset().
4. loadRunProgram() checks the passed program pointer (not shown). If no program is specified, the FEP is left in a reset state. If a program is specified, then the *fepManager* extracts the mailbox and ring-buffer addresses from the program header and passes them to the FEP's I/O manager, *fepIo*.setIoAddresses().
5. setIoAddresses() then maps the FEP addresses into the shared memory region of the Back End Processor, using *fepDevice*.mapAddress().
6. loadRunProgram() then calls loadSections() to copy all code and data sections, mapped to shared memory regions, into the FEP's memory.

7. `loadSections`() iterates through each program section, using *`fepDevice`*`.mapAddress`() to determine if a section is mapped to shared memory, and if so, to convert the FEP address into the corresponding BEP's shared memory address. `loadSections`() and then uses *`mongoose`*`.copyWords`() (not shown) to copy the section's code and/or data directly into the FEP memory.
8. Once all shared memory sections have been loaded, `loadRunProgram`() calls *`fepDevice`*`.releaseReset`() to allow the FEP to execute the partially loaded code (NOTE: This assumes that the program contains a section loaded at the Reset Vector). At this point, the *`fepManager`* assumes that the code running on the FEP is capable of handling Write Memory, and Execute Memory requests, sent via the FEP's Command MailBox.
9. `loadRunProgram`() then calls `loadSections`() again, this time specifying that it should load only sections which are not located contained in shared memory.
10. `loadSections`() iterates through the program sections, using *`fepDevice`*`.mapAddress`() to determine if a section is not contained in shared memory (not shown). It then calls the **`FepManager`**'s `writeMemory`() function which uses the Command Mailbox to instruct the FEP to load the contents of these sections into its memory. `writeMemory`() accesses the FEP's Command Mailbox using the I/O manager's `issueCmd`() member function.
11. Once all of the sections have been loaded, `loadRunProgram`() extracts the start execution address from the program header, and uses `executeMemory`() to cause the FEP to execute from the start of its completely loaded program. `executeMemory`() uses the I/O Manager's `issueCmd`() function to send the request (not shown). At this point, the FEP is up and running the loaded program.

25.5.3 Use 3: Read FEP Memory

Figure 108 illustrates the steps invoked in reading a section of a FEP's memory.

FIGURE 108. Read FEP Memory



1. *client* requests the contents of a FEP's memory, issuing *fepManager*.readMemory(), which blocks until the request is satisfied, or until an error is detected.
2. readMemory() determines if the requested region is within shared memory using *fepDevice*.mapAddress(), and if so, passes the supplied address to *mongoose*.copyWords() (not shown) to copy the region directly out of the FEP, and returns.
3. If the requested region is not contained within the FEP's shared memory region, readMemory() check that the request does not cross an instruction cache boundary within the FEP, and returns an error of so (not shown). If the requested region does not cross an instruction cache boundary, readMemory() enters a loop to copy the region out in sections, using the FEP's Command Mailbox. The loop uses *mongoose*.isIcache() (not shown) to determine if the mailbox request is for a read from memory, or a read from instruction cache.The loop then determines the amount of data that can be read by one command, using *fepIo*.getMaxCmdArgs().
4. readMemory() issues a request to the I/O manager to read a portion of the memory, using *fepIoManager*.issueCmd().
5. issueCmd() attempts to obtain exclusive access to the FEP's Command Mailbox, using waitForLock().
6. waitForLock() attempts to obtain the semaphore, using *lock*.waitFor().
7. waitForLock() ensures that the FEP is powered on, using *fepDevice*.hasPower()

8. `waitForLock`() ensure that the FEP is not in a reset or crashed state using *`fepDevice`*.`isReset`().
9. If `waitForLock`() is successful, `issueCmd`() copies the command information into the FEP's Command Mailbox, and sets its state to indicate that a new message is present (not shown). It then calls `waitForReply`() to wait until the FEP processes the command.
10. `waitForReply`() obtains a pointer to the currently running task, using *`taskMonitor`*.`queryCurrentTask`().
11. It then enters a polling loop, which terminates when either the FEP's Command Mailbox state no longer indicates that it contains a new message, until the loop's counter expires, or until the FEP loses power (`hasPower`()) or is reset (`isReset`()). Upon each iteration, the loop invokes **`Task`**::`sleep`() to allow other tasks to run. Meanwhile, the FEP periodically polls the mailbox. When it sees that a new message is ready, it executes the read-memory command, copying the requested memory region into the mailbox, and setting the box state indicating that a reply is ready (not shown). `waitForReply`() detects that the reply is ready, copies the FEP supplied memory contents into the caller's data buffer and returns.
12. Once `waitForReply`() returns, `issueCmd`() calls *`lock`*.`release`() to allow other tasks to use the FEP's Command Mailbox.
13. Upon each `issueCmd`() iteration, `readMemory`() calls `checkMonitor`() to ensure that the Task Monitor does not reset the instrument during a long memory read.
14. `checkMonitor`() uses *`taskManager`*.`queryCurrentTask`() to get a pointer to the currently running task.
15. `checkMonitor`() then uses **`Task`**::`requestEvent`() to test for a query from the monitor.
16. If a monitor request is present, `checkMonitor`() calls *`taskMonitor`*.`respond`() on behalf of the running task.

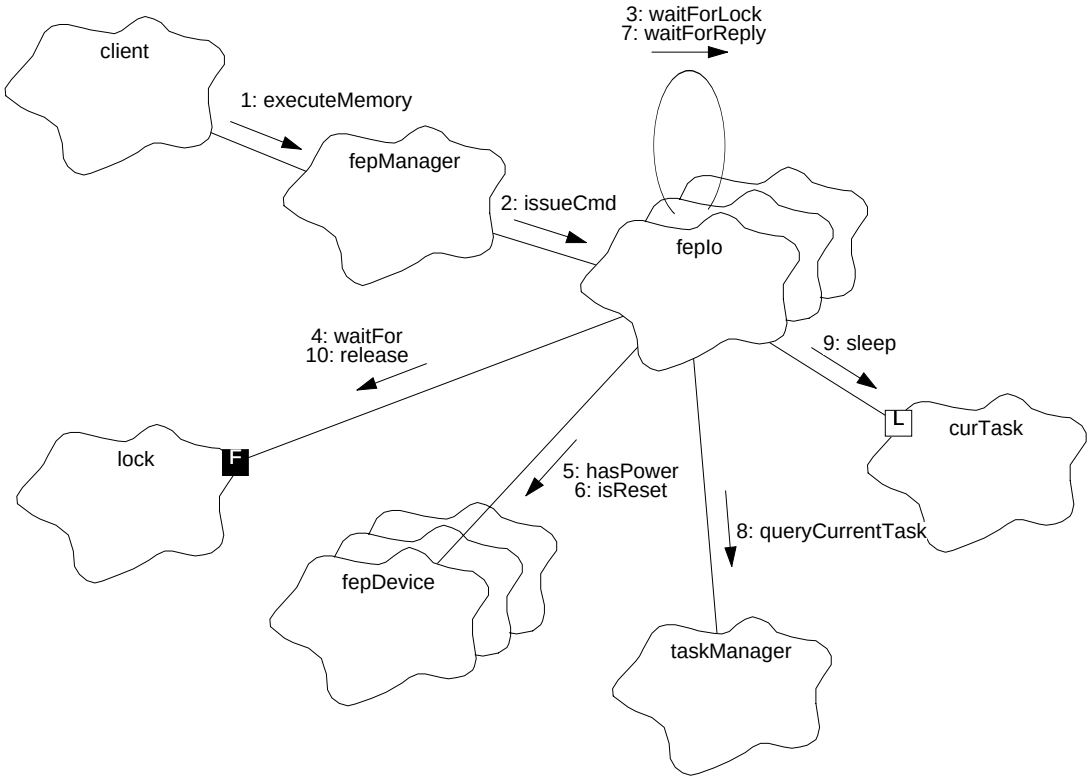
25.5.4 Use 4: Write FEP Memory

The scenario invoked in writing to the contents of a FEP's memory is very similar to that to **Use 3: Read FEP Memory**. Instead of calling `readMemory`(), the client calls *`fepManager`*.`writeMemory`(). `writeMemory`() then performs similar control operations to `readMemory`(), except that it uses *`mongoose`*.`copyWords`() to write the contents of FEP shared memory, and issues Write Memory Command Mailbox requests, rather than read requests, for writes to local memory regions.

25.5.5 Use 5: Execute FEP Subroutine

Figure 109 illustrates the steps involved in invoking a subroutine on a FEP.

FIGURE 109. Call FEP Subroutine



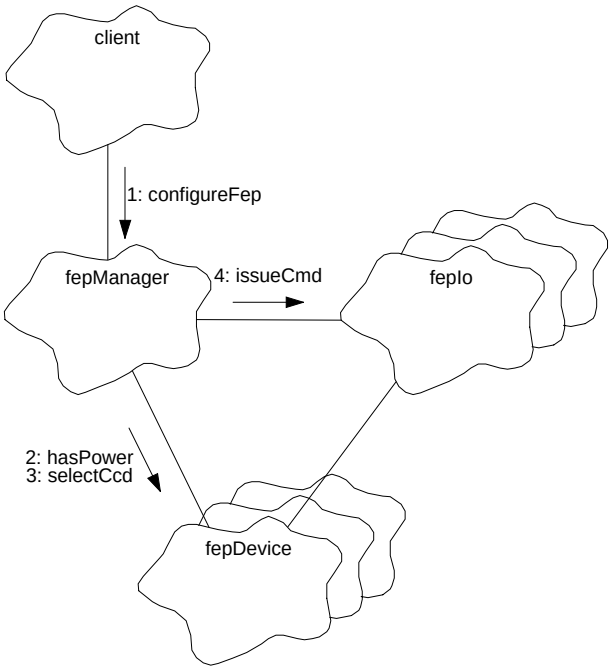
1. *client* tells the FEP Manager to invoke a subroutine on a particular Front End Processor, using *fepManager*.executeMemory().
2. executeMemory() forms and issues the request to the FEP, using *fepIo*.issueCmd().
3. issueCmd() attempts to obtain exclusive access to the FEP's Command Mailbox, using waitForLock().
4. waitForLock() waits for and obtains the mailbox's semaphore, using *lock*.waitFor().
5. Once *lock*.waitFor() returns, waitForLock() tests to ensure that the FEP has power using *fepDevice*.hasPower().
6. waitForLock() then checks to make sure the FEP has not been reset, using *fepDevice*.isReset().
7. Once the lock to a powered and running FEP is obtained, issueCmd() writes the request to the FEP's Command Mailbox, and sets the mailbox state indicating a new message (not shown). issueCmd() then waits for a response from the FEP using waitForReply().

8. `waitForReply`() gets a pointer to the currently running task, using *`taskManager`*.`queryCurrentTask`().
9. `waitForReply`() enters a polling loop, testing the mailbox state to detect when a reply has been written by the FEP. The loop terminates when either a response has been written, the FEP is powered off, the FEP is reset, or the loop's iteration counter expires. Within the body of the loop, `waitForReply`() calls `sleep`() (for a TBD number of BEP timer ticks) on the current task to allow other processes to run.
10. Meanwhile, the FEP eventually reads its Command Mailbox and calls the indicated subroutine. Once the subroutine returns, the FEP writes the return value of the function back into the mailbox, and sets its state to indicate that a reply is ready. Back on the BEP, `waitForReply`() detects the response, and returns to `issueCmd`(). `issueCmd`() then releases the mailbox, by calling *`lock`*.`release`(). `issueCmd`() then returns to the *`fepManager`*, which in turn, returns to its client.

25.5.6 Use 6: Configure a FEP for a science run

Figure 110 illustrates the steps used to prepare a Front End Processor to perform bias calibrations, or process CCD data.

FIGURE 110. Configure FEP

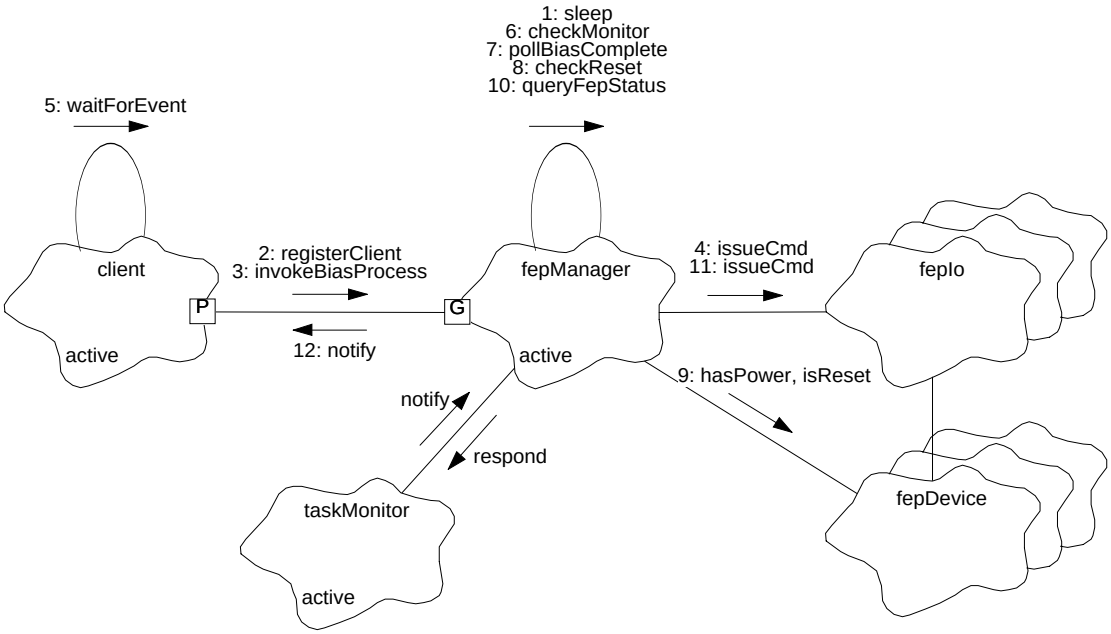


1. *client* configures a Front End Processor by calling *fepManager*.configureFep() indicating which CCD's is to be processed by the FEP and passing a parameter block to be used by the FEP.
2. Prior to setting the FEP's CCD selection in the FEP hardware, *fepManager*.configureFep() tests that the FEP is powered on, using *fepDevice*.hasPower().
3. configureFep() then tells the FEP hardware which CCD to listen to, using *fepDevice*.selectCcd().
4. configureFep() then issues a command to the FEP to load the parameter block, using *fepIo*.issueCmd() (see **Use 3: Read FEP Memory**, or **Use 5: Execute FEP Subroutine** for descriptions of the behavior of issueCmd()). If the load is successful, the FEP has registered the parameter block. configureFep() then adds the FEP to its list of enabled Front End Processors. Subsequent calls to invokeBiasProcess(), invokeDataProcess(), terminateProcess(), and readRecord() use this list to determine which FEPs to act on.

25.5.7 Use 7: Start bias calibrations

Figure 111 illustrates the steps used to start a bias calibration on all of the configured Front End Processors (see **Use 6: Configure a FEP for a science run**).

FIGURE 111. Start FEP Bias Calibrations



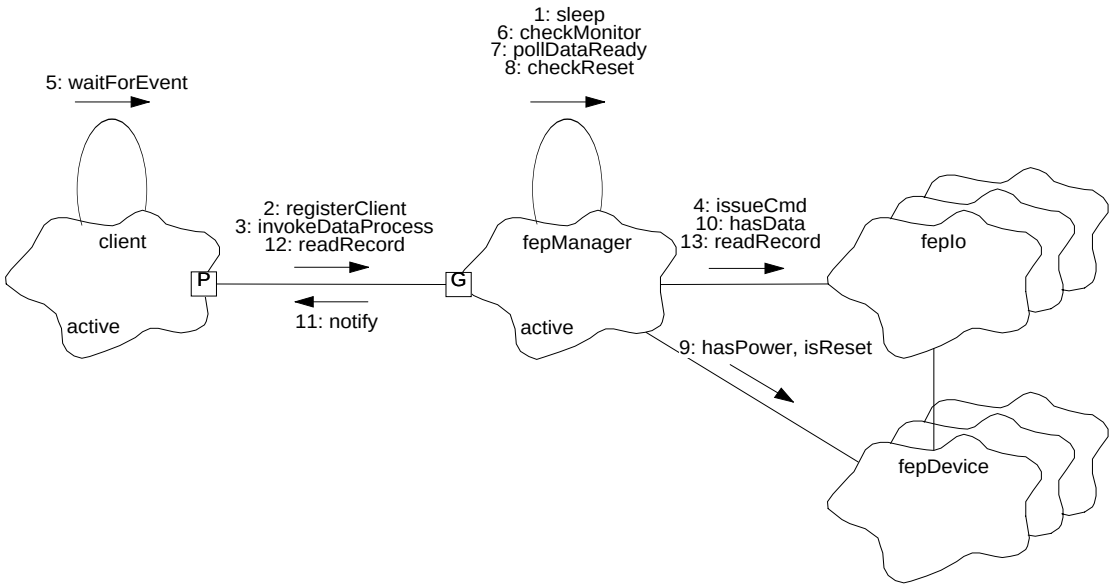
1. Upon each iteration of its main loop, the FEP Manager's main task loop (`goTaskEntry`) suspends execution using `sleep`(), allowing other tasks to run. Periodically, the task wakes up and checks for *`taskMonitor`* queries, FEP crashes, bias completions or data availability.
2. *`client`* informs the FEP Manager which task to notify, and with which events, when a bias calibration completes, when data becomes available, and if all of the configured FEPs should crash, using *`fepManager`*`.registerClient`().
3. *`client`* tells the FEP Manager to start bias calibrations on all of the configured FEPs, using `invokeBiasProcess`().
4. *`fepManager`*`.invokeBiasProcess`() loops through each configured FEP, using *`fepIo`*`.issueCmd`() to tell the FEP to start its bias calibration (see **Use 3: Read FEP Memory**, or **Use 5: Execute FEP Subroutine** for descriptions of the behavior of `issueCmd`()).
5. *`client`* then starts the DEA sequencer, recording the microsecond science time stamp associated with the start of the sequence (not shown). *`client`* then suspends until the bias calibration on all of the configured FEPs complete, until an error occurs, or until commanded to abort the current process, using `waitForEvent`().

6. Periodically, the *fepManager* task wakes up from its sleep and responds to any queries from the *taskMonitor*, using checkMonitor(). checkMonitor() responds to a pending query using *taskMonitor*.respond().
7. The fepManager task then detects that a bias computation is progress, and polls the active FEPs using pollBiasComplete().
8. pollBiasComplete() iterates through each enabled FEP, calling checkReset() to ensure that the FEP still has power and is not reset.
9. checkReset() uses *fepDevice*.hasPower(), and *fepDevice*.isReset() to ensure that the specified FEP is on and running. If a FEP is off-line, checkReset() takes the FEP out of the enabled list and pending bias list using disableFep() (not shown).
10. If the current FEP is on-line and is not yet in the bias completion list, pollBiasComplete() uses queryFepStatus() to query the current state of the FEP. (NOTE: A single FEP can hold up the start of data processing by computing bias "forever")
11. queryFepStatus() uses *fepIo*.issueCmd() to issue the query to the FEP, and retrieve the response. If the FEP's bias is complete, pollBiasComplete() adds the FEP to the completion list.
12. The *fepManager* task continues through its polling loop, repeatedly calling pollBiasComplete(). Once pollBiasComplete() indicates that all on-line FEPs have completed their bias calculations, the *fepManager* task notifies the client that the bias maps are complete, using *clientTask*->notify().

25.5.8 Use 8: Start data processing and consume data

Figure 112 illustrates the steps involved in starting science data processing on each of the configured Front End Processors (see **Use 6: Configure a FEP for a science run**).

FIGURE 112. Start FEP Data Processing



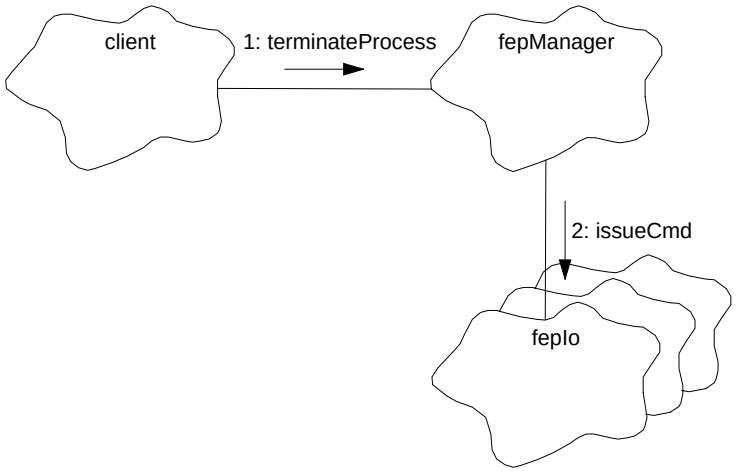
1. Upon each iteration of its main loop, the FEP Manager's main task loop (`goTaskEntry`) suspends execution using `sleep`(), allowing other tasks to run. Periodically, the task wakes up and checks for task monitor queries, FEP crashes, bias completions or data availability.
2. *`client`* informs the FEP Manager which task to notify, and with which events, when a bias calibration completes, when data becomes available, and if all of the configured FEPs should crash, using *`fepManager`*`.registerClient`().
3. *`client`* tells the FEP Manager to start data processing on all of the configured FEPs, using `invokeDataProcess`(), passing the command code indicating which science mode to run.
4. *`fepManager`*`.invokeDataProcess`() loops through each configured FEP, using *`fepIo`*`.issueCmd`() to tell the FEP to start its data processing activities (see **Use 3: Read FEP Memory**, or **Use 5: Execute FEP Subroutine** for descriptions of the behavior of `issueCmd`()).
5. *`client`* then starts the DEA sequencer, recording the microsecond science time stamp associated with the start of the sequence (not shown). *`client`* then suspends until data arrives, until an error occurs, or until commanded to abort the current process, using `waitForEvent`(). At this point, the CCDs start producing image data, and the FEPs start processing the received exposures. As a FEP produces data, it stores the data into its ring buffer.

6. Periodically `goTaskEntry`() wakes up from its call to `sleep`(), and responds to any taskMonitor queries using `checkMonitor`().
7. The task then detects that a run is in progress, and checks if a FEP has any data in its ring-buffer, using `pollDataReady`().
8. `pollDataReady`() iterates through each enabled FEP, calling `checkReset`() to ensure that the FEP still has power and is not reset.
9. `checkReset`() uses *`fepDevice`*.`hasPower`(), and *`fepDevice`*.`isReset`() to ensure that the specified FEP is on and running. If a FEP is off-line, `checkReset`() takes the FEP out of the enabled list using `disableFep`() (not shown).
10. If the current FEP is on-line, `pollDataReady`() uses *`fepIo`*.`hasData`() to check if the FEP has data in its ring buffer. If so, `pollDataReady`() stops checking the remaining FEPs and returns to its caller.
11. If a FEP has data in its ring-buffer, `goTaskEntry`() notifies the installed client task (*`clientTask`*) with the registered “data available” event, using *`clientTask`*->`notify`().
12. *`client`* wakes up from its call to `waitForEvent`() and calls *`fepManager`*.`readRecord`() to consume one science data record from one of the enabled FEPs.
13. *`fepManager`*.`readRecord`() iterates through each enabled FEP, calling *`fepIo`*.`readRecord`(). Once a *`fepIo`*.`readRecord`() indicates that a record has been read from its ring buffer, *`fepManager`*.`readRecord`() returns to its caller with the ring buffer data record, which FEP produced it, and which CCD the FEP is processing. The *`client`* then processes the record.

25.5.9 Use 9: Stop bias or data processing

Figure 113 illustrates the steps used to stop bias or science data processing on the configured Front End Processors. NOTE: Stopping science data processing is a normal part of performing a run, whereas, stopping a bias calibration process should only be used for error processing or under abnormal conditions.

FIGURE 113. Stop FEP Bias or Data Processing



1. *client* issues a request to stop the current operation (i.e. bias or data processing) using *fepManager*.terminateProcess().
2. terminateProcess() iterates through each configured Front End Processor, calling *fepIo*.issueCmd() to form and issue a "terminate" command via a FEP's Command Mailbox (see **Use 3: Read FEP Memory** or **Use 5: Execute FEP Subroutine** for a detailed description of issueCmd(). Once issueCmd() returns, the FEP software has received and acknowledged the command. At this point, the FEP completes its current operation (bias phase, or exposure) and returns to its top level control loop. If the FEPs were processing bias, the client is responsible for continuing to wait for indication that the bias completed (see **Use 7: Start bias calibrations**). If it was processing data, the client must continue to consume data from the FEPs until all of the FEPs have completed their exposures (i.e. client receives "end of exposure records", and the subsequent call to readRecord() indicate that there is no more data in the ring buffers).

25.5.10 Use 10: Disable FEP

In order to tell the FEP Manager to stop using a previously configured Front End Processor, the client calls *fepManager*.disableFep(). This takes the FEP out of the list of enabled FEPs, and subsequent calls to invokeBiasProcess(), invokeDataProcess(), terminateProcess(), and readRecord() no longer use the indicated FEP. In order to start using the FEP again, the client must call *fepManager*.configureFep().

25.5.11 Use 11: Write a bad pixel code into the pixel bias map and adjust its parity

In order to support the use of the FEP bias map to flag bad pixels, the FEP supports loading of bad pixel code directly into the FEP's pixel map (located in shared memory), and set the corresponding parity bit accordingly. To use this feature, the client calls *fepManager*.`loadBadPixel`(), specifying the FEP to address, and passing the pixel row and column of the bad pixel. `loadBadPixel`() then passes a PIXEL_BAD code to *fepIo*->`writeBiasValue`(). `writeBiasValue`() then computes the bias map address to write to and writes the code into the map. It then computes the parity of the passed code, and sets or clears the corresponding parity plane bit.

25.6 Class FepManager

Documentation:

This class is responsible for managing the Front End Processor devices.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **Task**

Implementation Uses:

```
FepDevice fepDevice[6]
FepIoManager fepIo[6]
TaskManager taskManager
TaskMonitor taskMonitor
```

Public Interface:

Operations:

```
FepManager()
configureFep()
disableFep()
executeMemory()
goTaskEntry()
invokeBiasProcess()
invokeDataProcess()
loadRunProgram()
powerOff()
queryFepStatus()
readMemory()
readRecord()
registerClient()
terminateProcess()
writeMemory()
```

Protected Interface:

Operations: `checkMonitor()`
`checkReset()`
`loadSections()`
`pollBiasComplete()`
`pollDataReady()`

Private Interface:

Has-A Relationships:

unsigned *enabledFeps*: This variable contains the list of FEPs currently configured to perform science operations. Bit 0 corresponds to FEP 0, bit 1 corresponds to FEP 1 and so on. If a bit is 1, then `configureFep()` has been called for the corresponding FEP, and the FEP has not been reset, powered off, or crashed. If a bit is 0, then the FEP was not configured, has crashed, or been powered off.

unsigned *expReadyFeps*: This variable contains the list of FEPs which have requested permission to proceed to the next exposure. Bit 0 corresponds to FEP 0, bit 1 to FEP 1 and so on. If a bit is 1, the corresponding FEP has issued a request since the last reply was issued by the BEP. If a bit is 0, the FEP is not yet ready to process another exposure.

unsigned *biasReadyFeps*: This variable indicates the list of FEPs who have completed their bias calibrations. Bit 0 corresponds to FEP 0, bit 1 to FEP 1, and so on. If a FEP has completed its bias, the corresponding bit in this value is 1. If not, the bit is 0.

unsigned *crashedFeps*: This variable contains a list of FEPs which caused an interrupt, and were in a reset state (i.e. their watchdog reset went off). Bit 0 corresponds to FEP 0, bit 1 to FEP 1 and so on. If a bit is 1, then the corresponding FEP has reset. If the bit is 0, the FEP has not autonomously reset (yet).

Task* *clientTask*: This is a pointer to the task using the **FepManager** to perform science operations. This variable is assigned using `registerClient()`.

unsigned *clientEvBiasRdy*: This is the event mask that the client task wants when all enabled FEPs have completed their bias calibrations. This value is assigned by `registerClient()`.

unsigned *clientEvDataRdy*: This is the event set that the client task wants if any of the enabled FEPs have new data in their ring-buffers. This value is assigned by `registerClient()`.

unsigned *clientEvCrash*: This variable contains the event mask that the client task wants if all of the enabled FEPs have watchdog reset. This variable is assigned by `registerClient()`.

`FepId` *`nxtFep`*: This variable indicates which FEP should be checked next by `readRecord`(). This variable is used to perform round-robin polling of each FEP's ring-buffer.

Concurrency: Active

Persistence: Persistent

25.6.1 FepManager()

Public member of: **FepManager**

Arguments:

unsigned *taskid*

Documentation:

This constructor initializes the **FepManager** instance, passing *taskid* to the parent **Task** constructor, zeroing its FEP lists, and client information, and registering the FEP interrupt callback instance, *fepCallback*, with the **FepIntrDevice**.

Concurrency: Sequential

25.6.2 checkMonitor()

Protected member of: **FepManager**

Return Class: **void**

Documentation:

This function responds to *taskMonitor* queries on behalf of the currently running task. It gets the current task, using *taskManager*.queryCurrentTask(), and then calls requestEvent() on that task for the query from the monitor. If the event is present, this function calls *taskMonitor*.respond(), to acknowledge the query.

Concurrency: Guarded

25.6.3 checkReset()

Protected member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function tests the FEP's, *fepid*, power and reset lines. If either the power is off, or if the FEP is held in a reset state, this function adds the FEP to the manager's crashed list, *crashedFeps*, and removes the FEP from the manager's enabled and ready lists (*enabledFeps*, *expReadyFeps*) by calling disableFep(). The function then returns *BoolTrue*. If the FEP has power and is not reset, this function returns *BoolFalse*.

Concurrency: Guarded

25.6.4 configureFep()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
CcdId *ccdselect*
unsigned* *pbaddr*
unsigned *pblen*

Documentation:

This function configures the Front End Processor indicated by *fepid*, and adds the FEP to the manager's list of enabled Front End Processors, *enabledFeps*. *ccdselect* is the CCD assigned to the configured FEP. *pbaddr* is a pointer to a FEP parameter block, and *pblen* is the number of words in the parameter block. This function loads the parameter block into the FEP via its Command Mailbox. If successful, this function returns *BoolTrue*. If the FEP is reset, powered off, or has a problem with a parameter, this function returns *BoolFalse*, and the FEP is not added to the enabled list.

Concurrency: Guarded

25.6.5 disableFep()

Public member of: **FepManager**

Return Class: **void**

Arguments:

FepId *fepid*

Documentation:

This function causes the FEP Manager to remove the Front End Processor, specified by *fepid*, from its enabled, bias ready, and data ready lists (*enabledFeps*, *biasReadyFeps*, *expReadyFeps*), and resulting in the FEP no longer reading data from its ring-buffer. Subsequent calls to `invokeBiasProcess`, `invokeDataProcess`, and `terminateProcess` commands have no effect on the specified FEP. In order to re-enable use of the FEP, the client must call `configureFep`() to re-add the FEP to the manager's enabled list.

Concurrency: Synchronous

25.6.6 executeMemory()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
unsigned (*)(...) *fepaddr*
const unsigned* *args*
unsigned *argcnt*
unsigned* *result*

Documentation:

This function tells the **FepManager** to command the Front End Processor, specified by *fepid*, to execute a subroutine, located at *fepaddr*. The subroutine will be passed at least *argcnt* words, whose values are contained in the buffer pointed to by *args*. Since, in C and C++, the caller maintains the argument stack, the implementation may pass more arguments than specified by the called function. The additional arguments are ignored by the called function. The value returned by the called subroutine will be stored in *result*. If the call is successful, this function returns *BoolTrue*, and result will contain the value returned by the subroutine. If an error is encountered, such as the indicated FEP is no power, or is in a reset state, this function returns *BoolFalse*, indicating that the call was not made.

Concurrency: Synchronous

25.6.7 goTaskEntry()

Public member of: **FepManager**

Return Class: **void**

Documentation:

This function is the main loop of the FEP Manager task. This loop is responsible for handling low-priority requests from one or more of the FEPs. This function consists of an infinite loop, which sleeps for 0.1 second on each iteration, after which it tests for event notifications from `serviceDevice`() and from the *`taskMonitor`* using `requestEvent`(). If the *`taskMonitor`* queries the manager, this function responds to the query. If `serviceDevice`() notifies the task that a FEP has crashed or that an FEP has requested service, this function checks the list of enabled FEPs. If the last enabled FEP has crashed, and a client task has been registered, this function notifies the task that there are no remaining enabled FEPs. If a FEP has requested service, this function tests each of the enabled FEPs output mailboxes and ring buffers, and notifies the registered client task if all enabled FEPs have completed their bias calibrations, or if any of the enabled FEPs have data ready in their ring buffers.

Concurrency: Synchronous

25.6.8 invokeBiasProcess()

Public member of: **FepManager**

Return Class: **Boolean**

Documentation:

This function instructs the FEP Manager to tell each of the configured Front End Processors (see `configureFep`()) to start computing bias map values from their respective CCDs. This function returns *`BoolTrue`* if successful, and *`BoolFalse`* if an error is encountered.

Concurrency: Guarded

25.6.9 invokeDataProcess()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

int *requestType*
int *acktype*

Documentation:

This function tells each of the configured Front End Processors (see `configureFep`()) to start processing data according to the request code, specified by *`requestType`*. If successful, and all of the FEPs respond with *`acktype`*, this function will return *`BoolTrue`*. If an error is encountered, this function returns *`BoolFalse`*. NOTE: This function is provided as a means by which new science modes can be added without having to directly modify the **FepManager**.

Concurrency: Guarded

25.6.10 loadRunProgram()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
const FepProgram* *program*

Documentation:

This function enables power to the Front End Processor, indicated by *fepid*, sleeps for 1 timer tick (100ms), loads the code and data referenced by *program*, and starts the FEP running the loaded program. If the indicated FEP is already powered on, its reset line is asserted prior to the load attempt. This function returns *BoolTrue* if program was loaded successfully, and *BoolFalse* if an error is encountered during the load.

Semantics:

Make sure power is on using *fepDevice*.powerOn(), sleep for 1 tick, and reset the processor using *fepDevice*.holdReset(). Extract the mailbox and ring buffer addresses from the program header and inform the I/O manager, *fepIo*.setIoAddresses(). Then call loadSections() to load sections into shared memory. If successful, release the reset line (using *fepDevice*.releaseReset()) to cause the FEP to execute its bootstrap loader code. Then use loadSections() to load the remaining sections via the command mailbox (serviced by the bootstrap loader code). Given that the command mailbox requires handshaking from the FEP, no explicit delays need to be inserted prior to this activity. Once all of the code is loaded, extract the start address and use executeMemory() to launch the completely loaded program.

Postconditions:

If successful, program is loaded into the indicated FEP's memory, and the FEP is running.

Concurrency: Guarded

25.6.11 loadSections()

Protected member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
const FepSection* *sections*
unsigned *sectioncnt*
Boolean *sharedOnly*

Documentation:

This function loads the code/data sections into the Front End Processor indicated by *fepid*. The argument *sections* points to the sections to load, and *sectioncnt* is the number of sections being pointed to. If *sharedOnly* is *BoolTrue*, only sections which appear in shared memory are loaded. If *sharedOnly* is *BoolFalse*, then this function uses the command mailbox to load sections into non-shared memory areas of the FEP.

Concurrency: Guarded

25.6.12 powerOff()

Public member of: **FepManager**

Return Class: **void**

Arguments:

FepId *fepid*

Documentation:

This functions turns the power off to the Front End Processor indicated by *fepid*, and takes the FEP out of the current list of enabled FEPs. In order to restart a powered-off FEP, use loadRunProgram().

Concurrency: Guarded

25.6.13 pollBiasComplete()

Protected member of: **FepManager**

Return Class: **Boolean**

Documentation:

This function queries the status of each active FEP to determine if their respective biases have completed. If so, the function returns *BoolTrue*. If one or more of the FEPs are continuing their bias computations, the function returns *BoolFalse*.

Concurrency: Guarded

25.6.14 pollDataReady()

Protected member of: **FepManager**

Return Class: **Boolean**

Documentation:

This function checks each enabled FEP's ring buffer. If any enabled FEP has data in its ring buffer, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Concurrency: Guarded

25.6.15 queryFepStatus()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
Boolean& *biasReady*
unsigned*& *biasbase*
unsigned*& *paritybase*

Documentation:

This function issues a query to the FEP to obtain its current status. fepid specifies which FEP to query. On return, *biasReady* will contain *BoolTrue* if the bias map has been computed and is ready for use and *BoolFalse* if the bias map has not been computed, or is in the process of being computed. *biasbase* will contain a pointer to the bias map memory buffer within the FEP shared memory (mapped to BEP address space), and *paritybase* will point to the bias map parity plane within the FEP shared memory (again, mapped to BEP address space). If the query succeeds, the function returns *BoolTrue*. If the query fails, the function returns *BoolFalse*.

Concurrency: Synchronous

25.6.16 readMemory()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
FepAddr *fepaddr*
unsigned* *dstbuf*
unsigned *wordcnt*

Documentation:

This function instructs the FEP Manager to read *wordcnt* 32-bit words from *fepaddr* on the FEP Device indicated by *fepid*. The read data is stored in *dstbuf*. This function returns *BoolTrue* if the read is successful, and *BoolFalse* if an error is encountered.

Concurrency: Synchronous

25.6.17 readRecord()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

RINGREC& *blockout*
FepId& *fepout*
CcdId& *ccdout*

Documentation:

This function consumes one block from one of the active Front End Processors ring buffers. If a block is available from one of the FEPs, this function copies the data into the passed output array, *blockout*, stores the id of the FEP producing the block into *fepout*, and stores the id of the CCD which produced the data into *ccdout*. It then returns *BoolTrue*. If no active FEP has data available, the function returns *BoolFalse*.

Concurrency: Guarded

25.6.18 registerClient()

Public member of: **FepManager**

Return Class: **void**

Arguments:

Task* *client*
unsigned *evbias*
unsigned *evdata*
unsigned *everr*

Documentation:

This function registers a client task using the **FepManager**. *client* points to the task to notify under various conditions. *evbias* is the event mask to use when a bias calibration completes on all enabled FEPs. *evdata* is the event mask to use when data is available on any of the enabled FEPs. *everr* is the event mask to use if all enabled FEPs watchdog reset.

Concurrency: Guarded

25.6.19 terminateProcess()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

Boolean *abortFlag*

Documentation:

This function instructs each of the configured Front End Processors (see configureFep()) to terminate their current science operations. If *abortFlag* is *BoolFalse*, the FEP is asked to complete its current data set before stopping. If *abortFlag* is *BoolTrue*, the FEPs are asked to stop immediately. This function returns *BoolTrue* is the request is successful, and *BoolFalse* if an error is encountered.

Concurrency: Guarded

25.6.20 writeMemory()

Public member of: **FepManager**

Return Class: **Boolean**

Arguments:

FepId *fepid*
FepAddr *fepaddr*
const unsigned* *srcbuf*
unsigned *wordcnt*

Documentation:

This function instructs the FEP Manager to write *wordcnt* 32-bit words from *srcaddr* to *fepaddr* on the Front End Processor indicated by *fepid*. If successful, this function returns *BoolTrue*. If an error is encountered, this function returns *BoolFalse*. NOTE: Although *fepaddr* is written to on the FEP, it is treated as read-only address while being used by the BEP.

Concurrency: Synchronous

25.7 Class FepIoManager

Documentation:

This class is responsible for managing a Front End Processor's command and request mailboxes, and its ring buffer.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **none**

Implementation Uses:

```
FepDevice fepDevice[]
TaskManager taskManager
Task
```

Public Interface:

Operations:

```
FepIoManager()
getMaxCmdArgs()
hasData()
issueCmd()
readRecord()
setBiasMapInfo()
setIoAddresses()
waitForLock()
waitForReply()
writeBiasValue()
```

Private Interface:

Has-A Relationships:

Semaphore *lock*: This semaphore instance is used to arbitrate access to the FEP's command mailbox.

const unsigned *lock_timeout*: This read-only field contains the number of BEP timer ticks (1/10 second) used to wait for access to the FEP's command mailbox.

const unsigned *reply_timeout*: This read-only field contains the approximate number of BEP timer ticks (1/10 second) used to wait for a reply to a command written to the FEP's command mailbox.

const FepId *fepId*: This field contains the identifier of the Front End Processor managed by this **FepIoManager** instance. This field is read-only, and is assigned during construction of the class instance.

CMD_MBOX* *cmdBox*: This field points to the Front End Processor's command mailbox, located in the FEP's shared memory area.

RINGBUF* *ringBuf*: This field points the Front End Processor's ring-buffer structure, located in the FEP's shared memory area.

unsigned* *biasMap*: This is a pointer to the bias map memory for the FEP.

unsigned* *parityPlane*: This is a pointer to the bias map parity error plane in the FEP's shared memory.

Concurrency: Guarded

Persistence: Persistent

25.7.1 FepIoManager()

Public member of: **FepIoManager**

Arguments:

FepId *fepid*
unsigned *semid*

Documentation:

This constructor creates an I/O manager instance associated with the Front End Processor specified by *fepid*. *semid* is the RTX semaphore identifier to use with the command mailbox's **Semaphore**, *lock*.

Concurrency: Sequential

25.7.2 getMaxCmdArgs()

Public member of: **FepIoManager**

Return Class: **unsigned**

Documentation:

This function returns the maximum number of arguments that can be passed or returned in the FEP's mailbox.

Concurrency: Synchronous

25.7.3 hasData()

Public member of: **FepIoManager**

Return Class: **Boolean**

Documentation:

This function determines if there is any data in the FEP's ring buffer. It returns *BoolTrue* if there is data ready, and *BoolFalse* if there is no data in the ring buffer.

Concurrency: Synchronous

25.7.4 issueCmd()

Public member of: **FepIoManager**

Return Class: **Boolean**

Arguments:

int *cmdtype*
const unsigned* *info*
unsigned *infocnt*
const unsigned* *args*
unsigned *argcnt*
int *replytype*
unsigned* *replybuf*
unsigned *replycnt*

Documentation:

This function issues a command to the FEP command mailbox and waits for a reply. *cmdtype* is the command type to use. If not 0, *info* is a pointer to a command-specific information data structure, and *infocnt* is the number of 32-bit words in the structure. If not 0, *args* is a pointer to an array of 32-bit argument words, and *argcnt* is the number of words in the array. *replytype* is the expected command response type. If not 0, *replybuf* is a pointer to where the function will store reply data, and *replycnt* is the maximum number of words that can be stored into *replybuf*. If successful, this function returns *BoolTrue*. If an error is encountered, or a reply doesn't match *replytype*, this function returns *BoolFalse*.

Semantics:

Call `waitForLock`() to obtain exclusive access to the command mailbox. If successful, store *cmdtype* into the mailbox. If *info* is not zero, copy the data structure to the mailbox and set the length. If *args* is not zero, append *args* to the mailbox and extend the length. Then set the mailbox state to indicate a new message. Call `waitForReply`() to obtain the response to the command. Once a reply is received (or an error occurs) release the lock semaphore, using *lock*.`release`().

Concurrency: Synchronous

25.7.5 readRecord()

Public member of: **FepIoManager**

Return Class: **Boolean**

Arguments:

RINGREC& *dstblock*

Documentation:

This function reads one data record block from the FEP's ring buffer into *dstblock*. If a block was available, it copies the block into *dstblock*, advances the read pointer in the ring buffer and returns *BoolTrue*. If no data is available in the ring buffer, the function returns *BoolFalse*.

Concurrency: Guarded

25.7.6 setBiasMapInfo()

Public member of: **FepIoManager**

Return Class: **void**

Arguments:

unsigned* *mapaddr*
unsigned* *parityaddr*

Documentation:

This function sets the bias map address, *biasaddr*, and bias map parity error plane address, *parityaddr*.

Concurrency: Guarded

25.7.7 setIoAddresses()

Public member of: **FepIoManager**

Return Class: **Boolean**

Arguments:

FepAddr *inbox*
FepAddr *ringbuf*

Documentation:

This function sets the mailbox and ring buffer addresses of this FEP instance. *inbox* is the FEP address of the FEP's command mailbox, *ringbuf* is the FEP's address of its ring-buffer. If all of the addresses fit into the FEPs shared memory space, this function returns *BoolTrue*. If one or more of the addresses are not within shared memory, this function returns *BoolFalse*.

Concurrency: Guarded

25.7.8 waitForLock()

Public member of: **FepIoManager**

Return Class: **Boolean**

Documentation:

This function attempts to obtain exclusive access to the instance's semaphore, using *lock*.waitFor(). If successful, and the processor has power and is not reset (*fepDevice*[*fepId*]->hasPower(), *fepDevice*[*fepId*]->isReset()) this function returns *BoolTrue*. If it times out, or the processor is disabled, this function returns *BoolFalse*.

Concurrency: Synchronous

25.7.9 waitForReply()

Public member of: **FepIoManager**

Return Class: **Boolean**

Arguments:

int *replytype*
unsigned* *dstbuf*
unsigned *dstcnt*

Documentation:

This function polls the command mailbox until a reply is written by the FEP. If successful the function returns *BoolTrue*, and if *dstbuf* is not 0, this function copies at most *dstcnt* words from the reply buffer to *dstbuf*. If *dstbuf* is 0, no reply data is copied. If the reply times out, or if an error is encountered, this function returns *BoolFalse*.

Semantics:

Query *taskManager* for current task. Enter loop which terminates when the command mailbox state is no longer indicates a new message, or when the loop iterates *reply_timeout* times, or if the FEP loses power or is reset. On each iteration, call *sleep*() on the current task for 1 tick. Once the loop completes, test the mailbox state, and *replytype*. If all is well, and *dstbuf* is not zero, copy the reply data to *dstbuf*. Once the copy is complete, set the mailbox state to indicate that the box is available for use.

Concurrency: Guarded

25.7.10 writeBiasValue()

Public member of: **FepIoManager**

Return Class: **void**

Arguments:

unsigned *row*
unsigned *col*
unsigned short *value*

Documentation:

This function writes *value* to the bias map location indexed by *row* and *col*, and adjusts the corresponding parity bit accordingly.

Concurrency: Guarded

26.0 DEA Management Classes (36-53218 B)

26.1 Purpose

The purpose of the Detector Electronics Assembly (DEA) Management classes are to manage access to the DEA CCD-controller and system boards.

26.2 Uses

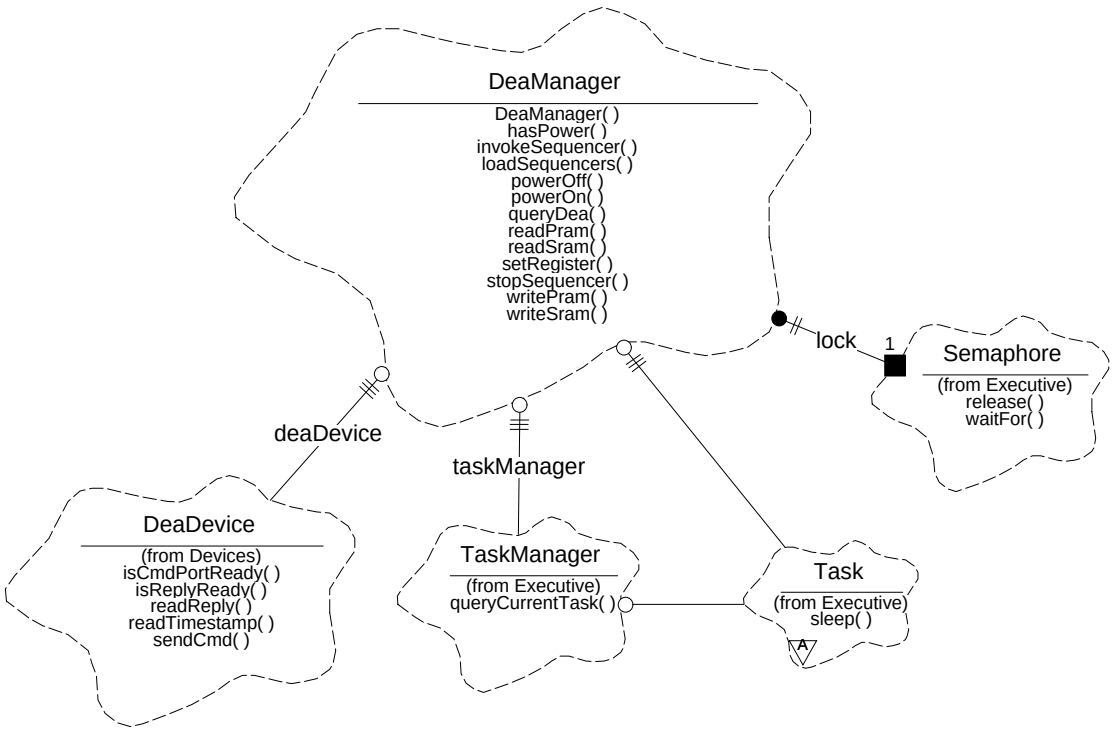
The following lists the use of the DEA Manager:

Use 1:: Load values into a CCD-controller's Program or Sequencer RAM
Use 2:: Read values stored into a CCD-controller's Program or Sequencer RAM
Use 3:: Start the CCD-controller sequencers
Use 4:: Stop the CCD-controller sequencers
Use 5:: Set the value of one of the DEA Registers or Digital-to-Analog converters
Use 6:: Obtain the current value of one of the DEA Registers
Use 7:: Enable and disable power to a single DEA CCD-controller board

26.3 Organization

Figure 114 illustrates the class relationships used by the Detector Electronics Assembly Manager class, **`DeaManager`**.

FIGURE 114. DeaManager Class Relationships



`DeaManager` - The **`DeaManager`** class is responsible for managing access to the Detector Electronics Assembly. It provides functions which read and write the contents of a CCD Controller board's Program and Sequencer RAM (`readPram`, `readSram`, `writePram`, `writeSram`), and to load complete images into the controller's RAM (`loadSequencers`). It provides functions to set control and digital-to-analog register values in one of the DEA boards and to query the values in the control and housekeeping registers (`setRegister`, `queryDea`). It provides functions to start and stop all of the DEA sequencers (`invokeSequencer`, `stopSequencer`). It also provides functions to individually power CCD-controller boards on and off and to determine if it has enabled power to a particular board (`powerOn`, `powerOff`, `hasPower`).

`Semaphore` - This class is supplied by the ***`Executive`*** class category. This class represents a resource lock or flag. Its implementation uses the underlying resource facilities of the RTX. The **`DeaManager`** contains an instance of this class, called *`lock`*, and uses this class to arbitrate access to the DEA interface. Prior to using the interface, the **`DeaManager`** attempts wait for exclusive access to the interface and reserve the semaphore (`waitFor`). Once the **`DeaManager`** has completed its action, it releases the semaphore (`release`).

`DeaDevice` - This class is supplied by the ***`Devices`*** class category. It is responsible for managing the physical interface between the Back End Processor and the Detector Electronics Assembly. The **`DeaManager`** uses this class to issue commands to the DEA and to retrieve status back from the DEA. This class provides functions which reset the DEA's interface board (`reset`() not shown), which indicate whether or not the command port to the DEA is ready to accept another command (`isCmdPortReady`), whether or not a status word has been received from the DEA (`isReplyRead`). It provides functions to write a command to be sent the DEA (`sendCmd`) and read status information sent back (`readReply`). The DeaDevice also provides a function which reads and returns the version of the science time-stamp, latched when the last command was issued to the DEA (`readTimestamp`).

`TaskManager` - This class is supplied by the ***`Executive`*** class category. It is responsible for managing the collection of tasks running within the Back End Processor. The **`DeaManager`** uses this class to obtain a pointer to the currently active task (`queryCurrentTask`).

`Task` - This class is supplied by the ***`Executive`*** class category. It represents and controls an active running task. The **`DeaManager`** uses this class to cause the current task to relinquish control to other tasks for a period of time (`sleep`).

26.4 DEA Manager Design Issues

26.4.1 Command Timing

This section describes how much time the DEA Manager must allow for commands to be transmitted to the DEA, and how much time it must allow for commands to be executed by the DEA. When issuing back-to-back commands, the DEA Manager must wait until the first command has been transmitted to the DEA and executed by the DEA before issuing a second command. If it sends the second command too close to the first, the DEA will ignore the second. Except for commands which demand a response from the DEA, there is no physical handshaking between the Back End Processor and the DEA. The command spacing requirements must be handled using time-delays.

The Back End Processor sends 24-bit commands to the DEA via a 1Mbps serial interface. It takes approximately 24us after the Back End writes a command word to the serial interface and the time at which it has been received by the DEA.

The time taken to execute a command varies from command to command. At a minimum, each command (including board and address selection) takes the DEA CCD Controllers 5us to execute.

The following tables list the various command types and their respective execution times:

TABLE 24. DEA CCD Controller Command Timing

Command	Register Type	Execution Time	Reply Time	Total Time (including transmission)
Select Card	N/A	9µs	N/A	33µs
Write Address	N/A	9µs	N/A	33µs
Write Data	PRAM/SRAM	9µs	N/A	33µs
Write Data	A/D Control Register	9.6ms	N/A	9.6ms
Write Data	Sequencer Control	19µs	N/A	43µs
Read Data	PRAM/SRAM	14µs	24µs	62µs
Read Data	Housekeeping	56.5µs	45.5µs	121µs
Read Data	Control Register	9µs	45.5µs	78.5µs

TABLE 25. DEA Interface Board Command Timing

Command	Execution Time	Reply Timing	Total Time (including transmission)
Read CCD Controller Housekeeping	56.6µs	45.5µs	121µs
Read Interface Housekeeping	56.6µs	45.5µs	121µs
Write Focal Plane Temperature	9µs	N/A	33µs
Write Back-out Temperature	9µs	N/A	33µs

TABLE 25. DEA Interface Board Command Timing

Command	Execution Time	Reply Timing	Total Time (including transmission)
Enable Relay	1.024ms	N/A	1.048ms
Switch CCD Controller Power	9µs[a]	N/A	33µs
Calibrate A/D Converter	9.6ms	N/A	9.6ms
Reset CCD Controllers	100ms	N/A	100ms

a. The CCD Controller power enable execution time does not include the board charge time. The instrument software will wait at 1/2 second after switching power on or off to a CCD Controller to allow the board to fully charge.

Assuming that the bulk of commanding to the DEA is to load Program and Sequencer RAM (PRAM and SRAM), the DEA Manager takes three approaches to command timing, fast commanding, and slow commanding. For commands which select boards, addresses and load RAM, the DEA Manager attempts to issue the commands as quickly as possible. For other commands which do not cause a response from the DEA, the DEA Manager adds the maximum delay between each command. For commands which request a response from the DEA, the DEA Manager blocks until the response has been received.

26.4.2 DEA CCD Controller Commanding

26.4.2.1 Command/Status Format and Overall Memory Layout

Refer to the “DPA/DEA Interface Control Document,” MIT 36-02205 Rev. A for descriptions of the command and status formats, and the overall address map of the DEA CCD Controller boards. This version of the document also accurately lists the D/A Controller settings.

26.4.2.2 Control Register Formats

The following illustrates the 8-bit DEA CCD Controller Control Registers. Unless otherwise specified, all single-bit signals are active high (1 - enable, 0 - disable):

Address	(msb) 7	6	5	4	3	2	1	(lsb) 0
0x10000	100KHz Sequencer Offset (0..63)						Stop Sequencer	Start Sequencer
0x10001	100KHz Video ADC Offset (0..63)						Stop Video ADC	Start Video ADC
0x10002	-	Hold House-keeping Address	-	-	Video Output Enable Channel D	Video Output Enable Channel C	Video Output Enable Channel B	Video Output Enable Channel A
0x10003	-	-	-	Status Enable (active low)	High Speed Tap Enable (active low)	Clock Swap Enable	Back Junction Diode Enable	-

NOTE: In order to ensure that all the sequencers start at the same time, ACIS always broadcasts the “Start Sequencer/Stop Sequencer” command word to all controllers being used for a run. This means that all of the controllers must use the same 100KHz Sequencer Offset value.

26.4.2.3 CCD Controller Housekeeping Channel Assignments

The following lists the CCD Controller Housekeeping Channels:

Index	Description	Min. Value	Max. Value	Conversion
0	Parallel Image Array Voltage +	0	TBD	Volts = TBD(value)
1	Parallel Image Array Voltage -	0	TBD	Volts = TBD(value)

Index	Description	Min. Value	Max. Value	Conversion
2	Parallel Framestore Voltage +	0	TBD	Volts = TBD(value)
3	Parallel Framestore Voltage -	0	TBD	Volts = TBD(value)
4	Serial Output Register Voltage +	0	TBD	Volts = TBD(value)
5	Serial Output Register Voltage -	0	TBD	Volts = TBD(value)
6	Reset Gate Voltage +	0	TBD	Volts = TBD(value)
7	Reset Gate Voltage -	0	TBD	Volts = TBD(value)
8	Output Gate Voltage	0	TBD	Volts = TBD(value)
9	Scupper Voltage	0	TBD	Volts = TBD(value)
10	Reset Diode Voltage	0	TBD	Volts = TBD(value)
11	Drain Output Channel A Voltage	0	TBD	Volts = TBD(value)
12	Drain Output Channel B Voltage	0	TBD	Volts = TBD(value)
13	Drain Output Channel C Voltage	0	TBD	Volts = TBD(value)
14	Drain Output Channel D Voltage	0	TBD	Volts = TBD(value)
15	RTD4 - Board Temperature	0	TBD	^{O}C = TBD(value)
16	RTD3 - SRAM Temperature	0	TBD	^{O}C = TBD(value)
17	RTD2 - ADC Temperature	0	TBD	^{O}C = TBD(value)
18	RTD1 - Gate Array Temperature	0	TBD	^{O}C = TBD(value)

26.4.2.4 Commanding Procedures

The DEA CCD Controllers are commanded using a small collection of control commands which can read and write a set of indexed control registers, D/A Converter settings, PRAM and SRAM memory locations, and A/D Converter housekeeping channels. The following describe the overall procedure for performing the indicated operation:

Writing to a Control Register, PRAM/SRAM or D/A Converter

1. Ensure that the desired card is selected, and issue a “Card Selection” command if not.
2. Use the “Write Address” command to select the destination register
3. Use the “Write Data” command to write the data value to the register and wait for the command to execute.

Reading a Control Register, or PRAM/SRAM

1. Ensure that the desired card is selected and is setup to reply to read requests. If not, ensure that the previously selected card’s “Status Enable” is disabled (use “Write Address”/”Write Data” to deselect), and use “Card Selection” to select the new card, and then assert “Status Enable” (using “Write Address”/”Write Data”) on the newly selected card.
2. Use the “Write Address” command to select the control register or PRAM/SRAM location.

3. Use the “Read Data” command to issue the query.
4. Wait for the status word to arrive

Reading a housekeeping A/D Converter channel

Although the housekeeper query is being addressed to a CCD Controller card, the A/D converter on the DEA Interface card will be used to sample and transmit the signal back to the BEP. For this reason, all CCD Controller cards must have their “Status Enable” signals deasserted.

1. Ensure that the desired card is selected and setup to reply (without “Status Enable”), and that no card has its “Status Enable” signal asserted. If this or another card left “Status Enable” asserted, then deassert it (use “Card Selection,” if needed, followed by “Write Address”/”Write Data” to deassert the signal).
2. Use the “Write Address” command to select the A/D channel to sample
3. Use the “Read Data” command to issue the query.
4. Wait for the status word to arrive from the interface card.

26.4.3 DEA Interface Card Commanding

26.4.3.1 Command/Status Format

All commands to the DEA Interface card have the following overall format:

23	22	21	20	19	18 17	16	15 12	11 0
0	0	1	1	Execute = 0 Read = 1	Unused	Off = 0 On = 1	Task Number	Task dependent data

26.4.3.2 Task Numbers and Data Fields

The task numbers and data fields on the DEA Interface board are as follows:

Action	Task	Data Field											
		11	10	9	8	7	6	5	4	3	2	1	0
Read Interface Housekeeping	0xa	Channel Number (0 - 39 decimal)											
Read Focal Plane Temperature	0xb	-											
Read Relay Positions	0xc	-											
Set Focal Plane Temperature	0x1	12-bit temperature set-point											
Set Bakeout Tempearture	0x2	12-bit temperature set-point											
Set Spare D/A	0x3	Value											
Set CCD Controller Power	0x4	Board Power Selection Mask[a]											
Discrete Settings	0x5	-	-	-	Hold Hk. Addr.	-	-	-	-	-	-	Bake Out	LED
Pulse Actions	0x6	-	-	-	-	-	-	-	Reset CCD Controllers	-	-	-	Cal. A/D
Signal Selects	0x7	-	-	-	-	Disable Video Clock	Disable 100KHz Synch.	Disable Command Clock	Disable Command Data	-	-	-	Signal Path
Relay Enables	0x8	-	-	-	-	-	-	-	R5	R4	R3	R2	R1

a. Bit 0 corresponds to board 1, bit 1 corresponds to board 2 and so on.

26.4.3.3 Commanding Procedures

Unlike the DEA CCD Controllers, the DEA Interface Card does not rely on indexed registers for most of its functionalilty. Instead, it is commanded using a collection set of command opcodes, or “tasks.” Each control function or collection of control functions is explicitly indicated by a command task number and possibly an item selection bit-field with the task-dependent data.

Setting a value (such as focal plane temperature)

1. Form the command word with the appropriate task number, placing the data value in the task-dependent data portion of the command word, and set the “Execute/Read bit” to 0.
2. Send the command
3. Wait for the command to be sent and executed

Reading a value

1. Form the command word with the appropriate task number and setting the “Execute/ Read bit” to 1.
2. Issue the command
3. Wait for the status to be returned

Enabling a switch or state

1. Form the command word with the appropriate task number, set the “Off/On” bit to 1, and set a ‘1’ in the task-dependent data field for the items which are to be enabled.
2. Issue the command
3. Wait for the command to be executed

Disabling a switch or state

1. Form the command word with the appropriate task number, set the “Off/On” bit to 0, and set a ‘1’ in the task-dependent data field for the items which are to be disabled.
2. Issue the command
3. Wait for the command to be executed

Reading a Interface card housekeeping A/D converter channel

1. Form the command word with the appropriate task number, and place the Interface Card housekeeping item code in the task-dependent data field.
2. Issue the command
3. Wait for the status to be returned

Reading a CCD Controller housekeeping A/D converter channel

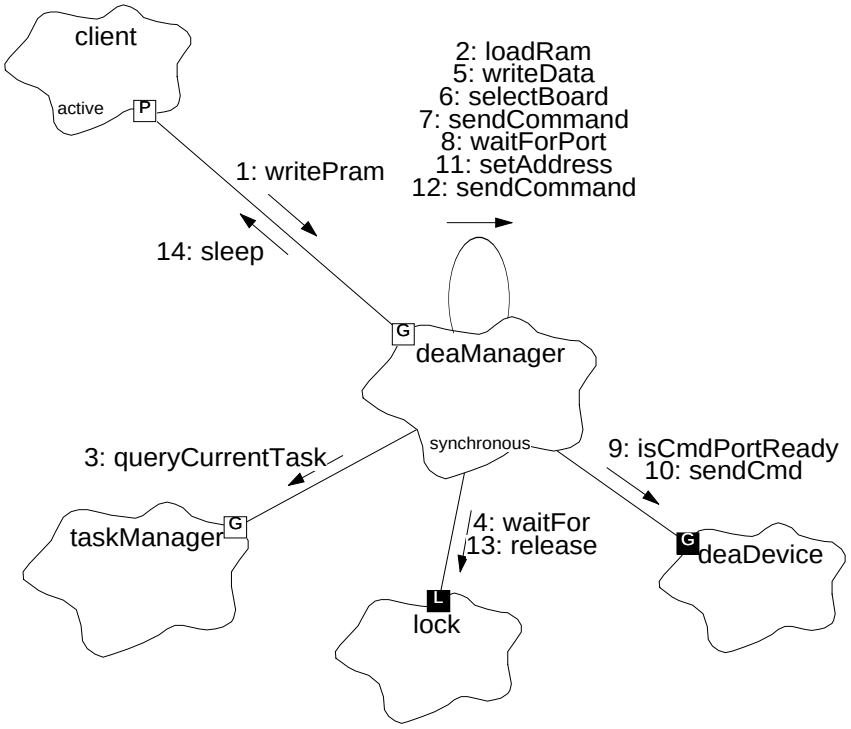
See Section 26.4.2.4

26.5 Scenarios

26.5.1 Use 1: Load Program or Sequencer RAM

Figure 115 illustrates the sequence of actions when a client instructs the **`DeaManager`** to load a section of Program RAM (PRAM). The scenario to load Sequencer RAM (SRAM) is the same except for the initial call to `writeSram`().

FIGURE 115. Load Program RAM Scenario



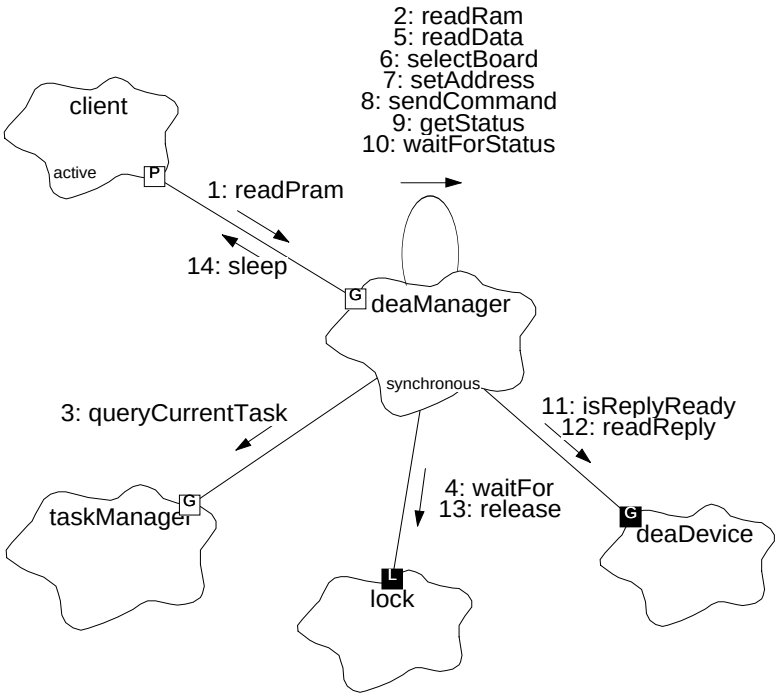
1. The *`client`* tells the *`deaManager`* to load a section of PRAM using *`deaManager`*.`writePram`().
2. The *`deaManager`* maps the requested PRAM address into an absolute DEA address, and calls `loadRam`() to perform the load.
3. `loadRam`() gets and saves a pointer to the running task using *`taskManager`*.`queryCurrentTask`().
4. `loadRam`() enters a loop where it loads each word into the DEA PRAM. The loop terminates once all words have been loaded, or on an error. The loop starts by symbolically obtaining exclusive access to the DEA by calling *`lock`*.`waitFor`().
5. `loadRam`() then checks an internal flag indicating if the sequencer is running or if the board is off (`hasPower`() not shown), and if so, releases the lock (*`lock`*.`release`() not shown) and returns an error. If the sequencer is not running, the `loadRam`() calls `writeData`() to write one word into the DEA board's RAM. For this scenario, assume that the sequencer is not running.

6. `writeData`() calls `selectBoard`() to ensure that the targeted DEA board has been selected.
7. `selectBoard`() checks the desired board against the currently selected board, and if different, issues a command to select the desired board using `sendCommand`(). For the purposes of this scenario, assume that `selectBoard`() needs to issue the command.
8. `sendCommand`() waits for the DEA command port to complete any transmissions in progress by calling `waitForPort`().
9. `waitForPort`() consists of a loop which polls the status of the DEA command port using *`deaDevice`*.`isCmdPortReady`(). If the loop's count expires, `waitForPort`() returns an error. If the command port finishes its transmission before the loop count terminates, `waitForPort`() returns that the port is ready. The loop count is chosen such that the time to exhaust the count is longer than the time to send one command to the DEA. For this scenario, assume that the previous command transmission completes.
10. Once the port is ready, `sendCommand`() performs a short busy-loop to allow the previous command to execute on the DEA, and then writes the new command to the DEA command port using *`deaDevice`*.`sendCmd`().
11. Once the desired board has been selected, `selectBoard`() returns to `writeData`(). `writeData`() then sets the destination address for the memory write using `setAddress`(). `setAddress`(), in turn, calls `sendCommand`() to issue the command (not shown).
12. `writeData`() then calls `sendCommand`() directly to write the data value into the DEA RAM, and then returns to `loadRam`().
13. `loadRam`()'s write loop then releases its hold on the DEA port, using *`lock`*.`release`().
14. `loadRam`() then checks how many words it has written since pausing to allow other tasks to run. If the number of words written exceeds the class-specific limit (defined by **`DeaManager`**::DEATIME_RAM_ITERATIONS), `loadRam`(), using its saved task pointer, tells the current task to sleep for a few tenths of a second, using *`client`*->`sleep`(). `loadRam`()'s loop then repeats from step 4 until all of the data words have been loaded into the DEA board's RAM.

26.5.2 Use 2: Read Program or Sequencer RAM

Figure 116 illustrates the sequence of actions when a client instructs the **`DeaManager`** to load a section of Program RAM (PRAM). The scenario to read Sequencer RAM (SRAM) is the same except for the initial call to `readSram`().

FIGURE 116. Read Program RAM Scenario



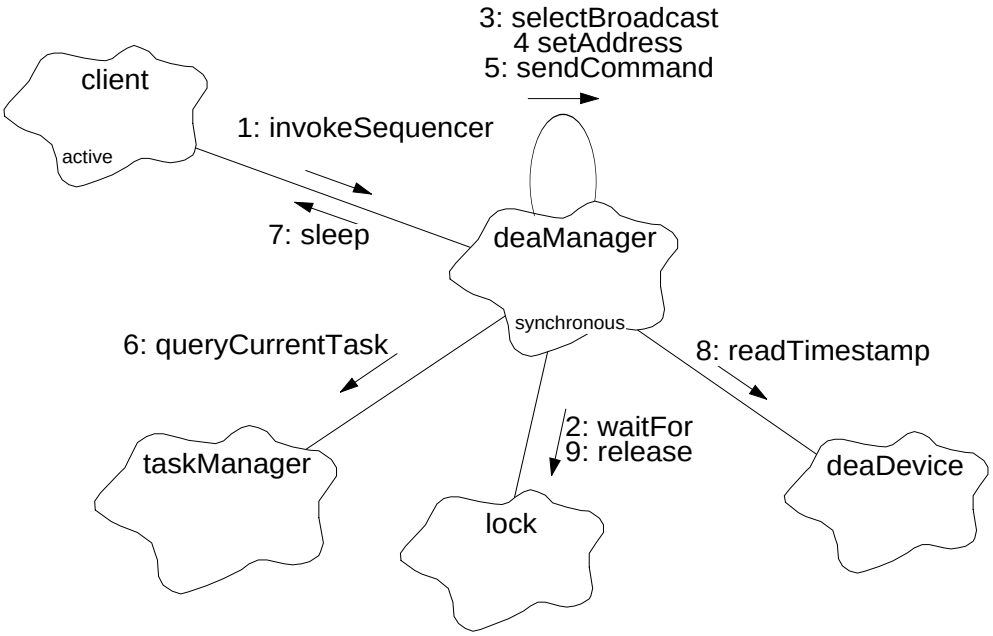
1. The *`client`* tells the *`deaManager`* to read section of PRAM using *`deaManager`*.`readPram`().
2. `readPram`() converts the SRAM index into a DEA address, and calls `readRam`() to perform the read.
3. `readRam`() obtains and saves a pointer to the currently running task, using *`taskManager`*.`queryCurrentTask`().
4. `readRam`() then enters its acquisition loop. The loop terminates once all of the requested words have been read, or when an error is encountered. The body of the loop starts by symbolically obtaining exclusive access to the DEA, using *`lock`*.`waitFor`().
5. `readRam`() then checks to see if sequencer is running or if the board is off. If so, it unlocks the DEA and returns an error. If not, it proceeds to read one word from the DEA using `readData`(). Assume for this scenario that the sequencer is not running.
6. `readData`() selects the desired board using `selectBoard`() (see Section 26.5.1 for more detail).
7. `readData`() then sets the desired read address using `setAddress`().

8. `readData`() then issues a data read command to the DEA using `sendCommand`().
9. Once the command has been issued, `readData`() obtains the returned data word using `getStatus`().
10. `getStatus`() waits for the returned value to arrive using `waitForStatus`().
11. `waitForStatus`() consists of a busy-loop which polls the DEA status port using *`deaDevice`*.`isReplyReady`(). If the loop's count expires before the status is returned, `waitForStatus`() returns an error. The busy-loop count is chosen to account for the execution time to read a value plus the transmission time of the value from the DEA to the BEP. Assume for this scenario that the reply arrives before the polling loop's count expires.
12. `getStatus`() then reads the status port using *`deaDevice`*.`readReply`(), passing the value back to its caller.
13. Once `readData`() has read and stored the RAM word into the destination buffer, `readRam`() releases the symbolic DEA lock using *`lock`*.`release`().
14. `readRam`() then checks how words it has written since pausing to allow other tasks to run. If the number of words read exceeds the class-specific limit (defined by **`DeaManager`**::DEATIME_RAM_ITERATIONS), `readRam`(), using its saved task pointer, tells the current task to sleep for a few tenths of a second, using *`client`*->`sleep`(). `readRam`()'s loop then repeats from step 4 until all of the data words have been read from the DEA board's RAM.

26.5.3 Use 3: Start the CCD-controller sequencers

Figure 117 illustrates the sequence of actions when a client instructs the **`DeaManager`** to start the CCD sequencers.

FIGURE 117. Start Sequencer Scenario

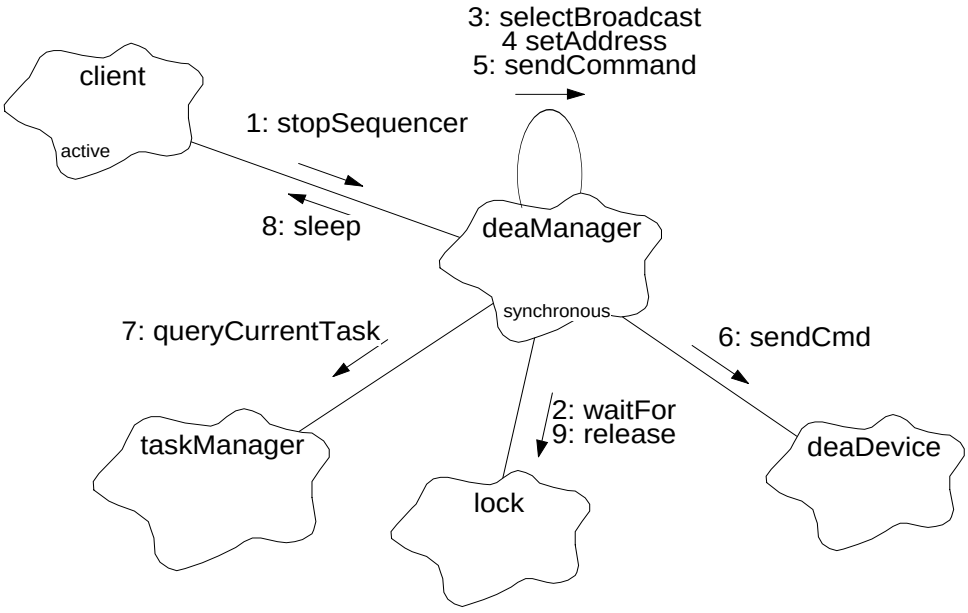


1. The *`client`* issues a command to start the DEA CCD sequencers, using *`deaManager`*.`invokeSequencer`().
2. `invokeSequencer`() obtains exclusive access to the DEA interface, using *`lock`*.`waitFor`().
3. `invokeSequencer`() enables all DEA CCD-controller boards to listen for commands using `selectBroadcast`().
4. `invokeSequencer`() selects the sequencer control register on all of the enabled boards using `setAddress`().
5. `invokeSequencer`() then forms and writes a sequencer enable word to the selected register on all of the enabled boards using `sendCommand`(). The sequencers will start clocking within 10us after the command is transmitted to the DEA.
6. `invokeSequencer`() obtains a pointer to the currently running task using *`taskManager`*.`queryCurrentTask`().
7. `invokeSequencer`() instructs the current task to sleep for a tenth of a second to ensure that the latched time-stamp is stable, using *`client`*->`sleep`().
8. `invokeSequencer`() then obtains a copy of the latched time-stamp using *`deaDevice`*.`readTimestamp`().
9. `invokeSequencer`() releases the lock, using *`lock`*.`release`() and passed the read time-stamp to the calling *`client`*.

26.5.4 Use 4: Stop the CCD-controller sequencers

Figure 118 illustrates the sequence of actions when a client instructs the **`DeaManager`** to halt the CCD sequencers.

FIGURE 118. Stop Sequencer Scenario

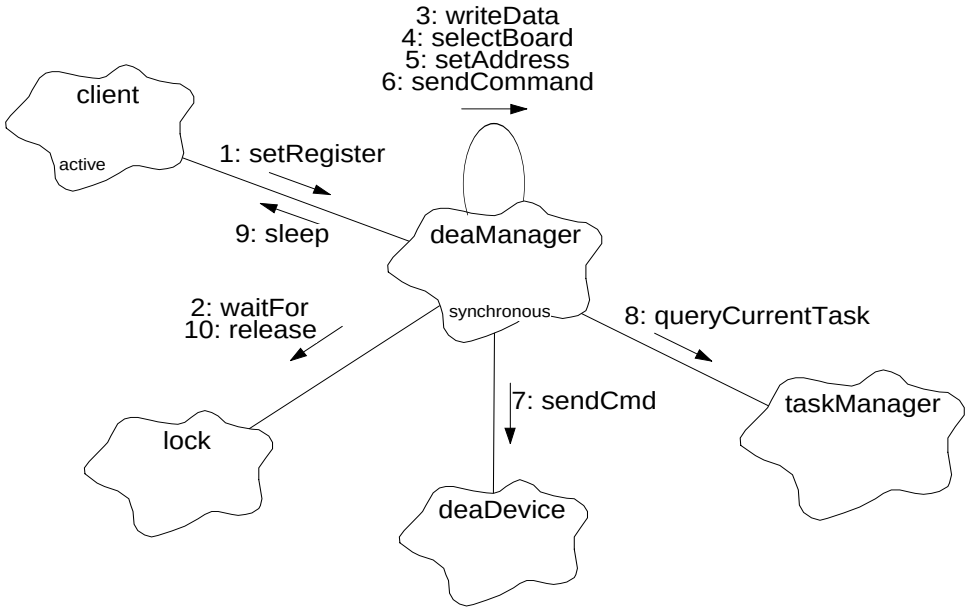


1. The *`client`* issues a command to halt the DEA CCD sequencers, using *`deaManager`*.`stopSequencer`().
2. `stopSequencer`() obtains exclusive access to the DEA interface, using *`lock`*.`waitFor`().
3. `stopSequencer`() enables all DEA CCD-controller boards to listen for commands using `selectBroadcast`().
4. `stopSequencer`() selects the sequencer control register on all of the enabled boards using `setAddress`().
5. `stopSequencer`() then forms and writes a sequencer disable word to the selected register on all of the enabled boards using `sendCommand`(). The sequencers will stop clocking once the command is transmitted to the DEA.
6. After ensuring that the command port is ready, `sendCommand`() issues the command to the DEA boards using *`deaDevice`*.`sendCmd`().
7. `stopSequencer`() obtains a pointer to the currently running task using *`taskManager`*.`queryCurrentTask`().
8. `stopSequencer`() instructs the current task to sleep for a tenth of a second to ensure that the command has been executed by the DEA, using *`client`*->`sleep`().
9. `stopSequencer`() releases the lock, using *`lock`*.`release`() and returns to the calling *`client`*.

26.5.5 Use 5: Set DEA Register

Figure 119 illustrates the sequence of actions when a client instructs the **`DeaManager`** to write a value to one of the DEA board registers.

FIGURE 119. Set DEA Register Scenario

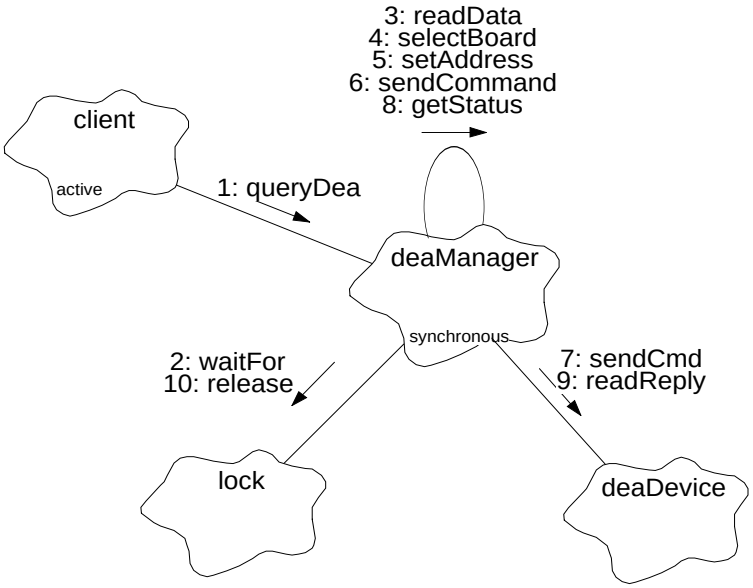


1. The *`client`* issues a command to write a value to one of the DEA boards, using *`deaManager`*.`setRegister`().
2. `setRegister`() obtains exclusive access to the DEA interface, using *`lock`*.`waitFor`(). It then checks if the board has power (`hasPower`() not shown), and if not, it releases the lock and returns an error.
3. `setRegister`() writes the data to the selected board register by calling `writeData`().
4. `writeData`() selects the desired DEA board using `selectBoard`().
5. `writeData`() selects the desired register using `setAddress`().
6. `writeData`() issues the command to write the data to the register using `sendCommand`().
7. After waiting for the command port to become available, `sendCommand`() transmits the command to the DEA using *`deaDevice`*.`sendCmd`().
8. Once the command has been issued, `setRegister`() gets a pointer to the currently running task using *`taskManager`*.`queryCurrentTask`().
9. `setRegister`() tells the task to sleep for several tenths of a second to ensure that the DEA has time to act on the register command using *`client`*->`sleep`().
10. `setRegister`() releases the lock using *`lock`*.`release`(), and returns to the calling *`client`*.

26.5.6 Use 6: Read DEA Register

Figure 120 illustrates the sequence of actions when a client instructs the **`DeaManager`** to read a value from one of the DEA board registers.

FIGURE 120. Query DEA Register Scenario

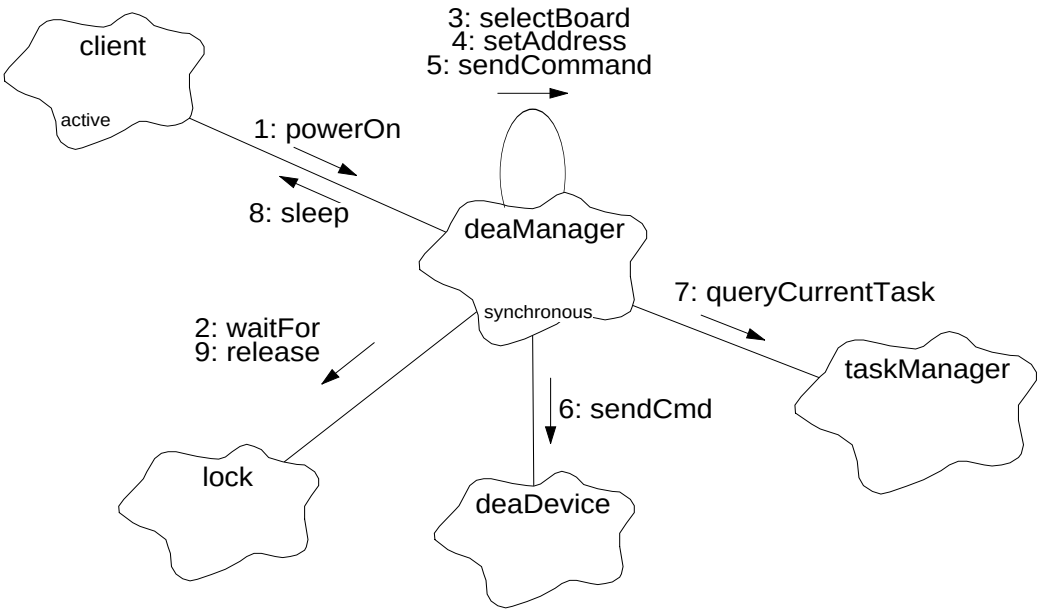


1. The *`client`* issues a command to read value from one of the DEA boards, using *`deaManager`*.`queryDea`().
2. `queryDea`() obtains exclusive access to the DEA interface, using *`lock`*.`waitFor`(). It then checks if the board has power (`hasPower`() not shown), and if not, it releases the lock and returns an error.
3. `queryDea`() reads the value from the desired board and register calling `readData`().
4. `readData`() selects the desired DEA board using `selectBoard`().
5. `readData`() selects the desired register using `setAddress`().
6. `readData`() issues the command to read the data from the register using `sendCommand`().
7. After waiting for the command port to become available, `sendCommand`() transmits the command to the DEA using *`deaDevice`*.`sendCmd`().
8. `readData`() then gets the response to the query using `getStatus`().
9. `getStatus`() waits for the reply to be received by the Back End, and reads the value using *`deaDevice`*.`readReply`().
10. `setRegister`() releases the lock using *`lock`*.`release`(), and returns to the calling *`client`*.

26.5.7 Use 7: Enable and disable power to a single DEA board

Figure 121 illustrates the sequence of actions when a client instructs the **`DeaManager`** to power on a single DEA CCD-controller board. The steps involved in powering off a single board are similar, except that a power-off command is sent to the DEA Interface board. TBD: How to detect and update system configurations on power-on?

FIGURE 121. CCD-Controller Power On Scenario



1. The *`client`* issues a command to enable power to one of the DEA CCD-controller boards, using *`deaManager`*.`powerOn`().
2. `powerOn`() obtains exclusive access to the DEA interface using *`lock`*.`waitFor`().
3. `powerOn`() issues a command to select the DEA Interface board on the DEA using `selectBoard`().
4. `powerOn`() issues a command to select the power-control register on the DEA Interface board using `setAddress`().
5. `powerOn`() adds the selected board to a mask of boards which have power, and issues a command to write the mask to the power control register using `sendCommand`().
6. `sendCommand`() waits for the command port to complete its previous transfer, and then issues the command to the DEA using *`deaDevice`*.`sendCmd`().
7. `powerOn`() obtains a pointer to the currently running task using *`taskManager`*.`queryCurrentTask`().
8. `powerOn`() instructs the task to sleep for 1 second (TBD) using *`client`*->`sleep`().
9. `powerOn`() releases its lock on the DEA interface using *`lock`*.`release`() and returns to the calling *`client`*.

26.6 Class DeaManager

Documentation:

The **`DeaManager`** class performs high level input/output operations to the Detector Electronics Assembly. It is capable of loading sequencer memory, setting register levels, and querying housekeeping ports.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`none`**

Implementation Uses:

`DeaDevice` *`deaDevice`*
`TaskManager` *`taskManager`*
`Task`

Public Interface:

Operations:

```
DeaManager()
hasPower()
invokeSequencer()
loadSequencers()
powerOff()
powerOn()
queryDea()
readPram()
readSram()
setRegister()
stopSequencer()
writePram()
writeSram()
```

Protected Interface:

Operations:

```
getStatus()
loadRam()
readData()
readRam()
selectBoard()
selectBroadcast()
sendCommand()
setAddress()
waitForPort()
waitForStatus()
writeData()
```

Private Interface:

Timing Constants:

DEATIME_RAM_ITERATIONS = 1000 per sleep
DEATIME_RAM_SLEEP = 2 ticks (0.2 seconds)
DEATIME_LOCK_WAIT = 5 ticks (0.5 seconds)
DEATIME_REG_SLEEP = 2 ticks (0.2 seconds)
DEATIME_SEQ_SLEEP = 2 ticks (0.2 seconds)
DEATIME_CMD_ITERATIONS = 100 (~50us)
DEATIME_CMD_WAITLOOP = 500 (~250us)
DEATIME_STAT_WAITLOOP = 1500 (~750us)
DEATIME_POWER_ON_SLEEP = 10 (1 second)
DEATIME_POWER_OFF_SLEEP = 10 (1 second)

Has-A Relationships:

`unsigned` *`curboard`*: This state variable indicates which DEA board was last written to and selected to send responses. If broadcast mode was last selected, or the last select command failed, this variable contains the value DEAID_UNKNOWN = 15.

`Boolean` *`sequencerActive`*: This instance variable indicates whether or not the sequencers have been commanded to run or not.

`Semaphore` *`lock`*: This is used to symbolically indicate exclusive access to the DEA interface.

`unsigned` *`powermask`*: This variable maintains a list of currently powered DEA Boards. Bit 0 corresponds to board 0, bit 1 to board 1, etc. If a bit is set to 1, a command has been sent to turn the power on the board. If a bit is set to 0, the board should be off.

Concurrency: Synchronous

Persistence: Persistent

26.6.1 DeaManager()

Public member of: **DeaManager**

Arguments:

unsigned *semid*

Documentation:

This is the constructor for the DEA Manager. *semid* is the Nucleus RTX semaphore id to use for the class's *lock* variable. The function initializes *lock*, sets *curboard* to DEAID_UNKNOWN, deasserts *sequencerActive*, and zeroes *powermask*. The body of the constructor calls *deaDevice*.reset() to reset the DEA interface board, and disable power to the DEA's CCD controller boards.

Concurrency: Sequential

26.6.2 getStatus()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned& *status*

Documentation:

This function busy-waits until a status is received from the DEA, and stores the read word into the status parameter. If the status word is read successfully, the function returns *BoolTrue*. If the wait expires before a status word is ready, the function returns *BoolFalse*.

Semantics:

Call waitForStatus() to synch. with the DEA. Then call *deaDevice*.readReply() to read the status word. If waitForStatus() times out, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

26.6.3 hasPower()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function returns whether or not the DEA Manager has enabled power to the board, indicated by *boardid*, by examining the private *powermask* variable. If the board has been enabled, the function returns *BoolTrue*. If not, it returns *BoolFalse*.

Concurrency: Guarded

26.6.4 invokeSequencer()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned& *starttime*

Documentation:

This function starts the DEA CCD Controller Sequencers. *starttime* is a copy of the science timestamp, latched when the command to start was issued. If the command is sent successfully, the function returns *BoolTrue*. If an error occurs, it returns *BoolFalse*.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Select all CCD-controllers as listeners by calling selectBroadcast(). Select the sequencer control register using setAddress() and form and issue the sequencer start command using sendCommand(). Obtain the calling task using *taskManager*.queryCurrentTask() and sleep for DEATIME_SEQ_SLEEP timer ticks (0.1 second) using *curtask*->sleep() to allow the command to complete execution. Read the latched time-stamp using *deaDevice*.readTimestamp() and place the result into *starttime*. Assert the *sequencerActive* flag, and release access to the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.5 loadRam()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
const unsigned short* *srcbuf*
unsigned *count*

Documentation:

This function loads *count* words pointed to by *srcbuf* into the location specified by *addr* on the board specified by *boardid*. If the load succeeds, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Call *taskManager*.queryCurrentTask() to get pointer to calling task. Set iteration counter to DEATIME_RAM_ITERATIONS and enter loop, which terminates when there are no more words to store, or on error. On each iteration, call *lock*.waitFor() to obtain exclusive access to the DEA. Then check *sequencerActive*, and generate an error if the sequencers are running. Call hasPower() to make sure board was turned on, and generate error if not. Call writeData() to store the word into DEA RAM and release the lock using *lock*.release() once the function returns. Then check the iteration counter and call *curtask*->sleep() and reset the counter if it has expired, otherwise, decrement the counter.

Concurrency: Guarded

26.6.6 loadSequencers()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

const DeaSequenceLoad& *image*

Documentation:

This function loads the sequencer images, specified by *image*, into the indicated CCD controller boards. If the load is successful, the function returns *BoolTrue*. If the load fails, the function returns *BoolFalse*.

Semantics:

Iterate through each section contained within *image*. On each section iteration, iterate through each selected controller board. If a board is selected, use writePram() to load PRAM sections into the boards, and writeSram() to load SRAM sections.

Concurrency: Guarded

26.6.7 powerOff()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function issues a command to turn off power to one of the DEA's CCD Controller boards. The function returns *BoolTrue* if successful, and *BoolFalse* if the command transmission fails.

Semantics:

Lock access to the DEA using *lock*.waitFor(). Call selectBoard() to select the DEA's interface board, and call setAddress() to select the power control register on that board. Clear the bit in *powermask* corresponding to *boardid*, and write *powermask* to the selected control register. Once the command has been issued, get a pointer to the current task using *taskManager*.queryCurrentTask(), and sleep for DEATIME_POWER_OFF_SLEEP using *curtask*->sleep(). Once the power-off time has elapsed, release control the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.8 powerOn()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function issues a command to enable power to the CCD controller board indicated by *boardid*. If the command is sent successfully, the function returns *BoolTrue*. If the command fails, it returns *BoolFalse*.

Semantics:

Lock access to the DEA using *lock*.waitFor(). Call selectBoard() to select the DEA's interface board, and call setAddress() to select the power control register on that board. Set the bit in *powermask* corresponding to *boardid*, and write *powermask* to the selected control register. Once the command has been issued, get a pointer to the current task using *taskManager*.queryCurrentTask(), and sleep for DEATIME_POWER_ON_SLEEP using *curtask*->sleep(). Once the power-on time has elapsed, release control the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.9 queryDea()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *queryid*
unsigned& *value*

Documentation:

This function queries the housekeeping port or PRAM location, addressed by *queryid*, on the DEA board indicated by *boardid*, and places the returned item into *value*. If the query command and response is successful, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Lock access to the DEA using *lock*.waitFor(). Call hasPower() to ensure that the desired board has been powered on. If the board is not on, return *BoolFalse*. Convert the *queryid* from a register index into a DEA address, and pass it, and the reference to *value* to readData(), which reads the requested register and places the result into *value*. Once the requested register has been read, release control the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.10 readData()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
unsigned& *value*

Documentation:

This function reads the field located at addr on DEA Board *boardid* and stores the result into value. If successful, the routine returns *BoolTrue*. If the read fails, it returns *BoolFalse*.

Semantics:

Call `selectBoard`() to select the board to read from. Call `setAddress`() to select the address to read. Call `sendCommand`() to issue the read command. Call `getStatus`() to read the response to the query.

Concurrency: Guarded

26.6.11 readPram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *pramid*
unsigned *index*
unsigned short* *dstbuf*
unsigned *valcnt*

Documentation:

This function reads *valcnt* words of PRAM from the board specified by *pramid*, and starting from the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a query fails, the function returns *BoolFalse*. This function adjusts index to the corresponding DEA address and calls readRam() to perform the read.

Concurrency: Guarded

26.6.12 readRam()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
unsigned short* *dstbuf*
unsigned *count*

Documentation:

This function reads *count* words from the DEA RAM located at *addr* on board *boardid* into the buffer *dstbuf*. If successful, the function returns *BoolTrue*, otherwise it returns *BoolFalse*.

Semantics:

Call *taskManager*.queryCurrentTask() to get pointer to calling task. Set iteration counter to DEATIME_RAM_ITERATIONS and enter loop, which terminates when there are no more words to read, or on error. On each iteration, call *lock*.waitFor() to obtain exclusive access to the DEA. Then check *sequencerActive*, and generate an error if the sequencers are running. Call hasPower() to make sure board was turned on, and generate error if not. Call readData() to fetch the word from DEA RAM. Release the lock using *lock*.release() once the function returns. Store the returned value into the destination buffer. Then check the iteration counter and call *curtask*->sleep() and reset the counter if it has expired, otherwise, decrement the counter.

Concurrency: Guarded

26.6.13 readSram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *sramid*
unsigned *index*
unsigned short* *dstbuf*
unsigned *valcnt*

Documentation:

This function reads *valcnt* words of DRAM from the board specified by *sramid*, and starting from the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a query fails, the function returns *BoolFalse*. This function adjusts index to the corresponding DEA address and calls readRam() to perform the read.

Concurrency: Guarded

26.6.14 selectBoard()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function selects *boardid* as the current board being written to and being queried. If the specified board has already been selected, or if the selection command succeeds, the function returns *BoolTrue*. Otherwise, the function returns *BoolFalse*.

Semantics:

Compare *boardid* to *curboard*. If different, form command, set the write mask bit corresponding to *boardid*, and the read select field to the passed *boardid* and issue the command using sendCommand(). If sendCommand() succeeds, copy *boardid* to *curboard* and return *BoolTrue*. If it fails, set *curboard* to DEAID_UNKNOWN and return *BoolFalse*.

Concurrency: Guarded

26.6.15 selectBroadcast()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function issues a command which selects all boards to listen to the next command. If the command is successful, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Semantics:

Set *curboard* to *powermask*, enabling all powered on boards as listeners, form command and set the read select to board DEAID_UNKNOWN. Issue the command using sendCommand(). If sendCommand() succeeds, return *BoolTrue*. If it fails, return *BoolFalse*.

Concurrency: Guarded

26.6.16 sendCommand()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned *command*

Documentation:

This function sends a command word, command, to the DEA. If the word is sent, the function returns *BoolTrue*. If the wait for the command port expires, the function returns *BoolFalse*.

Semantics:

Call waitForPort() to wait until the command port is ready. Then execute a small loop DEATIME_CMD_INTERATIONS times to ensure a delay between the previous command and the next. Then pass *command* to *deaDevice*.sendCmd(). If waitForPort() fails, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

26.6.17 setAddress()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

unsigned *address*

Documentation:

This function issues a command to copy *address* into current address register on the listening DEA board. If the command succeeds, the function returns *BoolTrue*. On error, it returns *BoolFalse*.

Semantics:

Form and issues the write address command using sendCommand(). If the command is sent successfully, the function returns *BoolTrue*. If the command fails, it returns *BoolFalse*.

Concurrency: Guarded

26.6.18 setRegister()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *regaddr*
unsigned *regval*

Documentation:

This function writes *regval* to the register located on the DEA board, *boardid*, at the register address, *regaddr*. This function returns *BoolTrue* if the command is successfully sent, and *BoolFalse* if it fails to send the value. After issuing the command, the function sleeps the calling task for at least 0.2 seconds to allow the command to be executed.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Check to ensure that the board has power, using hasPower(). Adjust the *regaddr* to map to the DEA register addresses and call writeData() to write the value to the specified board register. If it succeeds, get the current task, using *taskManager*.queryCurrentTask(), and sleep for DEATIME_REG_SLEEP (~0.2 seconds) to allow the command to be executed, release the lock using *lock*.release() and return *BoolTrue*. If the it fails, return *BoolFalse*.

Concurrency: Guarded

26.6.19 stopSequencer()

Public member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function issues a command to stop the DEA CCD Controller Sequencers. If the command is successful, the function returns *BoolTrue*. If an error is detected, the function returns *BoolFalse*.

Semantics:

Obtain exclusive access to the DEA interface using *lock*.waitFor(). Select all CCD-controllers as listeners by calling selectBroadcast(). Select the sequencer control register using setAddress() and form and issue the stop command using sendCommand(). Obtain the calling task using *taskManager*.queryCurrentTask() and sleep for DEATIME_SEQ_SLEEP timer ticks (0.1 second) using *curtask*->sleep() to allow the command to complete execution. Clear the *sequencerActive* flag, and release access to the DEA interface using *lock*.release().

Concurrency: Guarded

26.6.20 waitForPort()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function busy waits until the DEA serial command port is ready to send a word of data. If the wait time exceeds the serial transfer time (i.e. about 24us), the function returns *BoolFalse*, otherwise, it returns *BoolTrue*.

Semantics:

Iterate no more than DEATIME_CMD_WAITLOOP times, calling *deaDevice*.isCmdPortReady(). If the call returns *BoolTrue*, the port is ready for another command and return *BoolTrue*. Otherwise, continue iterating. If the loop iterations complete, the command port timed-out, and return *BoolFalse*.

Concurrency: Guarded

26.6.21 waitForStatus()

Protected member of: **DeaManager**

Return Class: **Boolean**

Documentation:

This function busy-waits until a status word is received by the DEA status port. If a status word is ready, the function returns *BoolTrue*. If the wait expires without a status word being received, the function returns *BoolFalse*.

Semantics:

Iterate DEATIME_STAT_WAITLOOP times or until *deaDevice*.isReplyReady() returns *BoolTrue*. If the device indicates a reply is ready, return *BoolTrue*. If the loop completes without a status word available, return *BoolFalse*.

Concurrency: Guarded

26.6.22 writeData()

Protected member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*
unsigned *addr*
unsigned *value*

Documentation:

This function writes *value* to the register located on the DEA board, *boardid*, at the register address, *addr*. This function returns *BoolTrue* if the command is successfully sent, and *BoolFalse* if it fails to send the value.

Semantics:

Call selectBoard() to ensure the appropriate board has been enabled. Then call setAddress() to select the register to write to, Then form and issue the write command using sendCommand() to send the value to the selected board. If all of the steps succeed, return *BoolTrue*, otherwise return *BoolFalse*.

Concurrency: Guarded

26.6.23 writePram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *pramid*
unsigned *index*
const unsigned short* *srcbuf*
unsigned *valcnt*

Documentation:

This function writes *valcnt* words of PRAM from the source buffer pointed to by *srcbuf*, into the board specified by *pramid*, and starting at the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a write fails, the function returns *BoolFalse*. This function maps *index* to the appropriate DEA address and uses loadRam() to perform the load.

Concurrency: Guarded

26.6.24 writeSram()

Public member of: **DeaManager**

Return Class: **Boolean**

Arguments:

DeaBoardId *sramid*
unsigned *index*
const unsigned short* *srcbuf*
unsigned *valcnt*

Documentation:

This function writes *valcnt* words of SRAM from the source buffer pointed to by *srcbuf*, into the board specified by *sramid*, and starting at the location indicated by *index* into the array pointed to by *dstbuf*. If successful, the function returns *BoolTrue*. If a write fails, the function returns *BoolFalse*. This function maps *index* to the appropriate DEA address and uses loadRam() to perform the load.

Concurrency: Guarded

27.0 Memory Server (36-53219 A)

27.1 Purpose

The **`MemoryServer`** responds to commands to read, write, and execute code in memory.

27.2 Uses

The MemoryServer will support requests to telemeter a copy of any portion of the memory's contents, to replace a section of the memories contents, or to execute the code stored in Instruction Cache or general purpose memory.

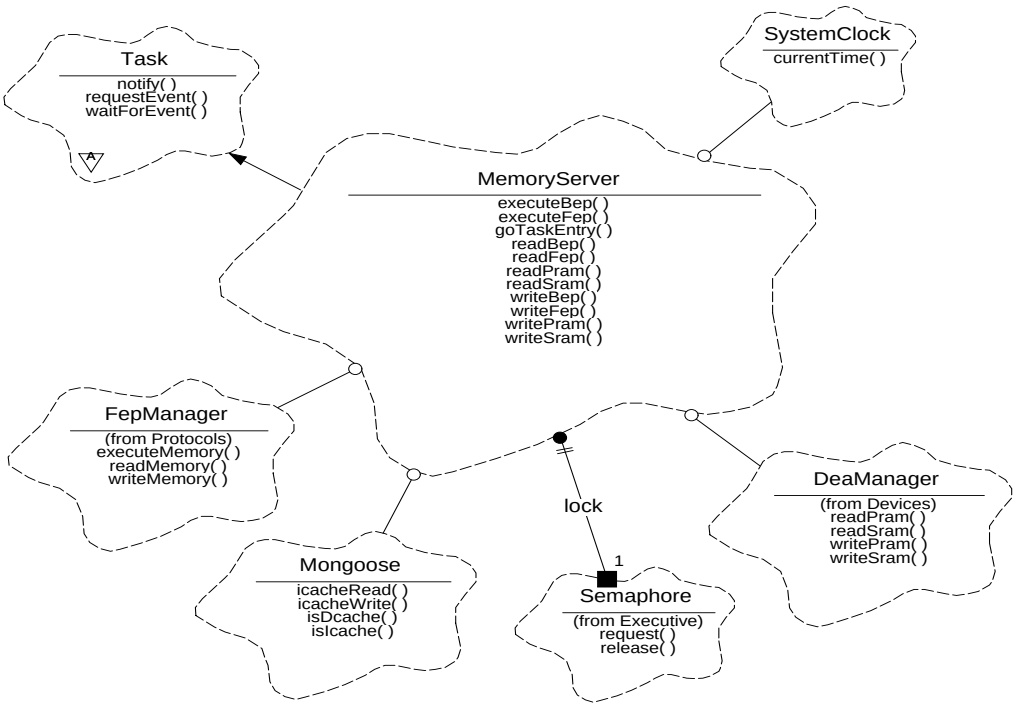
Specifically, it provides the following features:

Use 1:: Provides rapid acceptance of client requests through public routines
Use 2:: Waits responding to client requests and monitors' query
Use 3:: Reads memory of the BEP, FEP, PRAM, and SRAM
Use 4:: Writes memory of the BEP, FEP, PRAM, and SRAM
Use 5:: Executes appropriate BEP or FEP memory
Use 6:: Read and telemeter (i.e. dump) multi-packet configuration information (such as the Bad Pixel Map)

27.3 Organization

Figure 122 illustrates the relationships between the classes used by the MemoryServer, with public methods of interest listed; while Figure 2 shows the telemetry packet classes.

FIGURE 122. MemoryServer Class Relationships

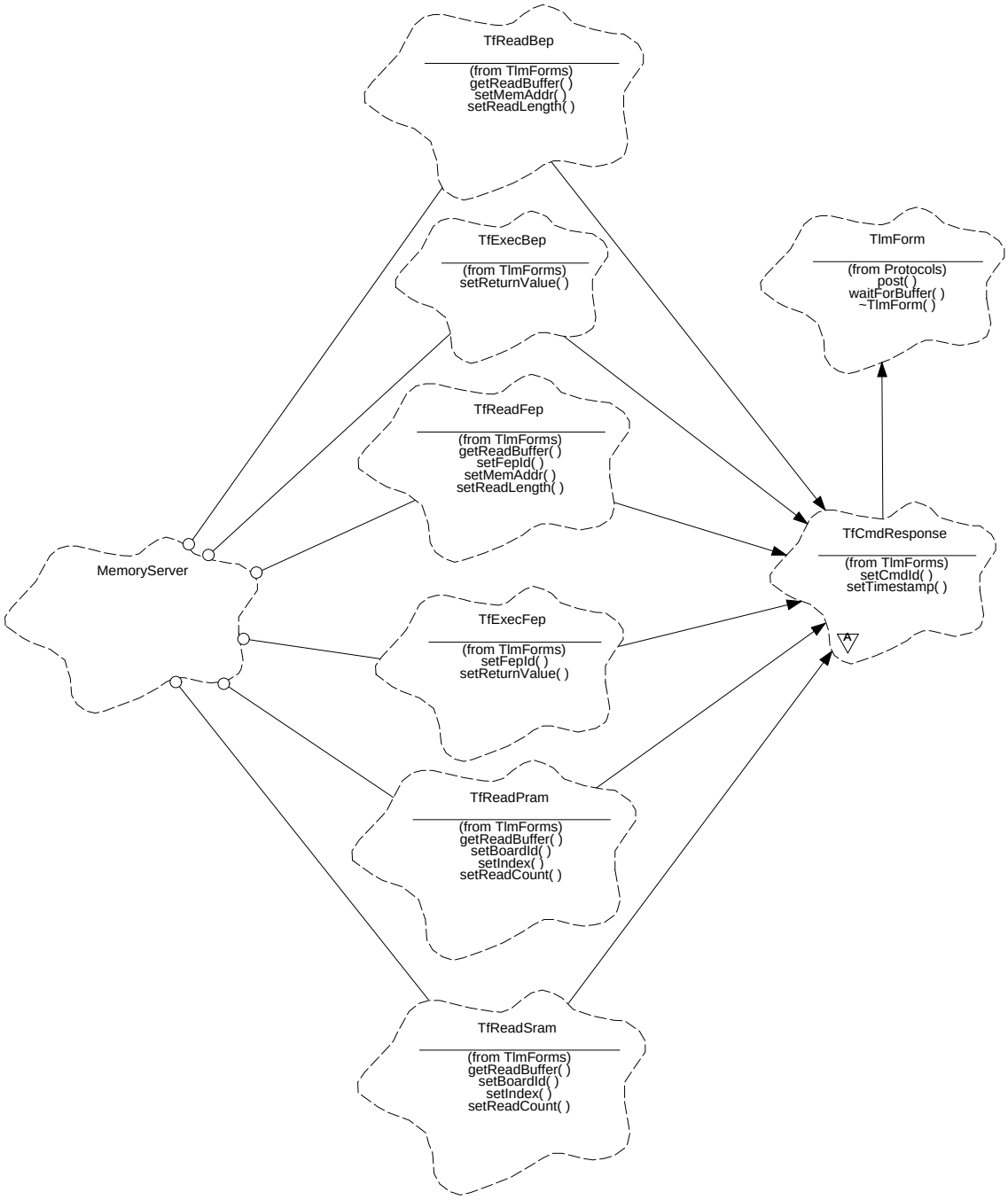


The MemoryServer uses the ***Executive*** and ***Protocols*** class categories.

MemoryServer - This class is a subclass of ***Executive::Task***. It is responsible for providing reads, writes, and executes of memory directly from the request.

TfReadBep - TfReadFep - TfReadSram - TfReadPram - TfExecBep - TfExecFep - This set of classes, Figure 123, encapsulates the representation of a telemetry packet. It is a subclass of ***Protocols::TfCmdResponse***, which in turn, is a subclass of ***TlmForm***.

FIGURE 123. MemoryServer Telemetry Packet Classes



Mongoose - This set of utility tools is available to all classes. The functions specific to the **`MemoryServer`** deal with reading and writing I_Cache.

SwHousekeeper - This class (not shown) is a subclass of ***`Executive::Task`***. It is responsible for delivering housekeeping data telemetry packets (faults, status, etc.) supplied to it by the various functions.

TaskMonitor - This class (not shown) is a subclass of ***`Executive::Task`***. It interrogates each thread in turn, verifying that it is functioning. Failure to respond will cause a watchdog reset when the watchdog counter reaches zero. The maximum time is less than eight minutes (TBD).

notify and **waitForEvent -** are functions of ***`Executive::Task`***. They provide the connectivity between the client's request and the thread's main process.

Semaphore - This class is a member of ***`Executive`***. It provides the gating method used to process commands serially.

FepManager - This class is a member of ***`Protocols`***. It is responsible for the read, write, and execute interaction with the FEPs.

DeaManager - This class is a member of ***`Protocols`***. It is responsible for the read, and write interaction with the DEAs.

27.4 Scenarios

27.4.1 Operational Overview

The MemoryServer has been designed to quickly acquire a client's request directive, and then release the client to its normal duty. The request will be fulfilled, eventually, by the MemoryServer task. All tasks must be able to respond to the task wellness monitor.

The MemoryServer is implemented by providing a set of public functions, callable by the client. Each of the set supports a specific type of request. It is the purpose of each function to verify arguments, to obtain a control semaphore which will insure serial execution of requests, and to quickly acquire buffered input data. The function will then notify the MemoryServer task, running autonomously, that a request is pending. Control is returned to the client.

When the MemoryServer task, `goTaskEntry`(), receives the notification of a pending request, it independently activates the private function responsible for fulfillment of that type of request. The fulfilling function, either directly or assisted by services provided by specialized managers, will complete the request. The function will then return to the MemoryServer task which releases the control semaphore, permitting additional requests to be initiated.

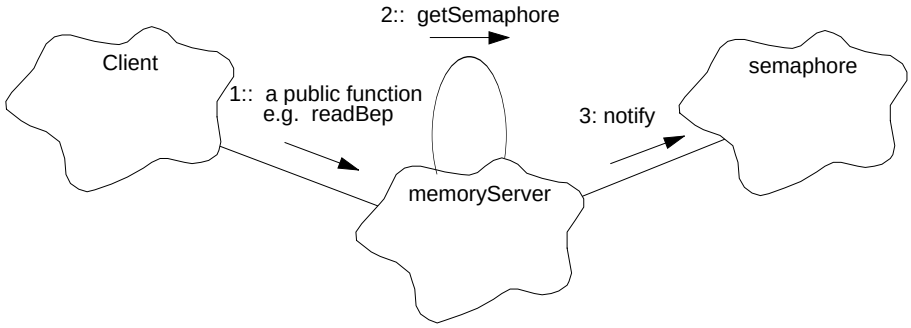
The acquiring public functions and their corresponding fulfilling private functions are: `executeBep` - `exeBep`, `executeFep` - `exeFep`, `readBep` - `rdBep`, `readFep` - `rdFep`, `readPram` - `rdPram`, `readSram` - `rdSram`, `writeBep` - `wrtBep`, `writeFep` - `wrtFep`, `writePram` - `wrtPram`, `writeSram` - `wrtSram`.

Between requests and during the request fulfillment period, the process must punctually respond to the task wellness monitor. Failure to do so will keep the monitor from resetting the watchdog.

27.4.2 Use 1:: Provides rapid acceptance of client requests through public routines

As illustrated in Figure 124, any client requesting a memory service will call one of the public routines with the arguments and data which specifies that request. The intent is to quickly initiate the request and return.

FIGURE 124. Public Function Delivers a Request

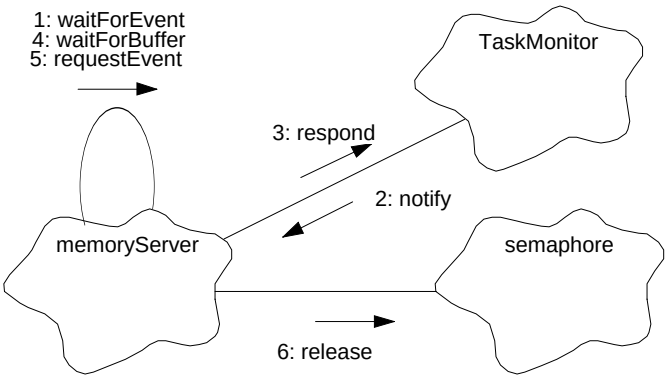


1. The client requests a memory service via a public function e.g. `readBep`(). Each service provides for verification of the arguments as necessary.
2. Then attempts to obtain control of the MemoryServer task using `getSemaphore`(). If the semaphore is already in use, the function returns BUSY. Upon obtaining the semaphore, the arguments and associated data are stored.
3. Finally, `notify`() is called to alert the fulfillment control task `goTaskEntry`(). The public function returns control to the client reporting the request as DELIVERED.

27.4.3 Use 2:: Waits responding to client requests and monitors' query

The MemoryServer task is omnipresent. It only functions when requested to do so by its public functions or by the monitor. Refer to Figure 125

FIGURE 125. Event handling by the MemoryServer



1. When there are no pending requests, the main MemoryServer process thread, `goTaskEntry`(), waits, idling in `waitForEvent`(), for a valid request from its public functions or for a query from the `taskMonitor`().
2. Meanwhile, the `taskMonitor`() is using **`Task`**`::notify()` to send a

EV_TASKQUERY to each active task to verify its viability.

3. On receipt of an event, `goTaskEntry`() uses **`TaskMonitor::`**`respond`() to answer the `taskMonitor`().
4. When an event is received from a Client driven MemoryServer public function requesting a memory service, `goTaskEntry`() directs the request to the proper subfunction for processing. Should this request fulfilling subfunction be delayed by the unavailability of packets to deliver requested data, it lingers for a specified period in `waitForBuffer`().
5. Each time it obtains a packet buffer and between lingerings, as necessary, the subfunction uses the non-blocking `requestEvent`() to monitor `taskMonitor`() queries. When a query does arrive, it uses `respond`() to send a reply to the `task-Monitor`().
6. Upon the functions return, the main process, `goTaskEvent`() calls **`sema-phore::`**`release`() to free the semaphore and then resumes its waiting.

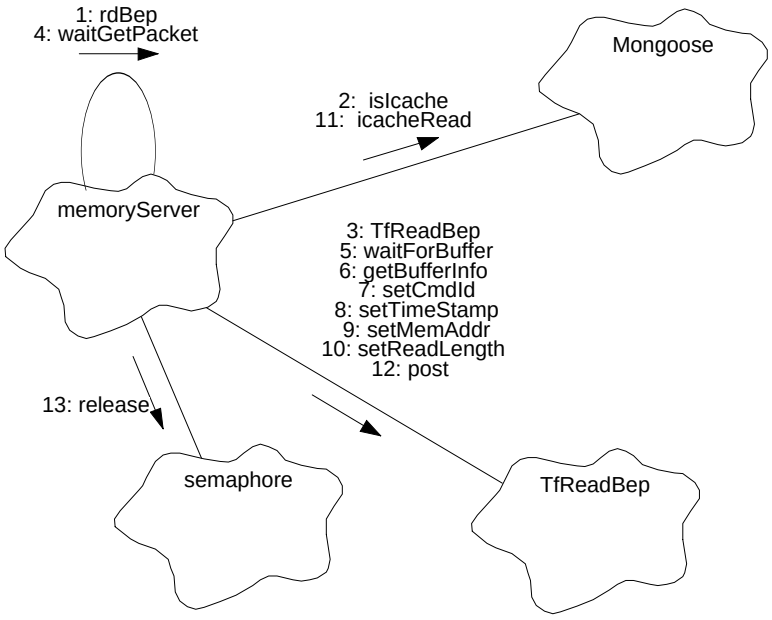
27.4.4 Use 3:: Read memory of the BEP, FEP, PRAM, and SRAM

The read requests will identify the target process memory, the memory location to begin the read and the amount of data to be delivered. A request to deliver an extensive amount of data will require an appreciable number of packet loads. With a limited supply of MemoryServer packets and possible contention for delivery services, this request may dribble data packets over an extended period.

27.4.4.1 Read Back End Processor memory

Figure 126 illustrates the process by describing the steps needed to deliver a copy of the Back End Processors' memory

FIGURE 126. Process a read BEP request



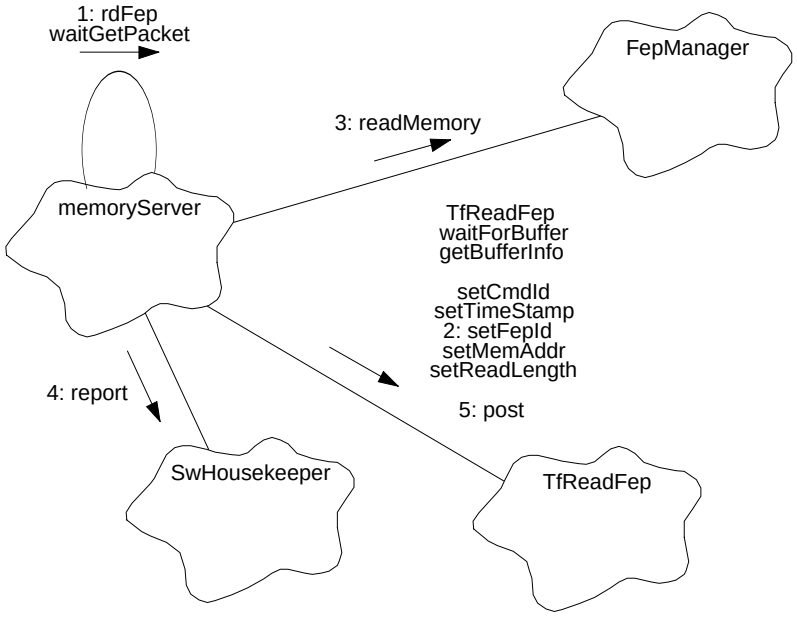
1. Upon being alerted by `notify`(), sent by the client through the public function readBep (not shown), the main process thread, `goTaskEvent`(), handles the request by directing the BEP reader service function, `rdBep`(), to process the request.
2. The `rdBep`() function uses `isIcache`() to test for an I_Cache read request so it may invoke the appropriate reading method.
3. It then enters a loop which uses the constructor to obtains a telemetry packet instance.
4. It calls the MemoryServer function `waitGetPacket`() to supply a packet buffer.
5. The routine checks the monitor (above) and uses the TfReadBep parent function **`TlmForm::`**`waitForBuffer`() which may linger for a stated period. It will return having succeeded in getting a buffer, or will cycle to try again.
6. `rdBep`() uses the forms' `getBufferInfo`() to determine the location of the buffer and its length.
7. Then it uses the forms' `setCmdId`() to install the command identifier into the packet buffer.
8. The forms' `setTimeStamp`() is used to provide a timing reference.
9. The address of the first word to be read and stored into the buffer is installed using the forms' `setMemAddr`(). The memory location of each packets' contents are thus identified.
10. Having determined the amount of data to be loaded in the current packet buffer, the forms `setReadLength`() installs this information.
11. `rdBep`() will call `icacheRead()` to have a copy of data in the I cache inserted into the buffer or else it will load the buffer directly itself.
12. A call to the forms' `post`() will forward the packet for delivery. `rdBep`() calculates the amount of data remaining to be processed. If the request has not been completed, it cycles to the top of the loop to obtain, load and deliver another packet.
13. When the request is completed, control is returned to the main process where the semaphore is made available to the next request using **`semaphore`**`::release`().

27.4.4.2 Read Front End Processor memory

Assistance from the FEP manager is required to deliver a memory copy from one of the FEPs. The BEP manages the packets needed to fulfill the request and will designate the packet buffer address to begin data storage. It, will specify which FEP memory is to be

read, where to begin and how much may be stored in the buffer. Figure 127 illustrates this shared activity.

FIGURE 127. Process a read FEP request



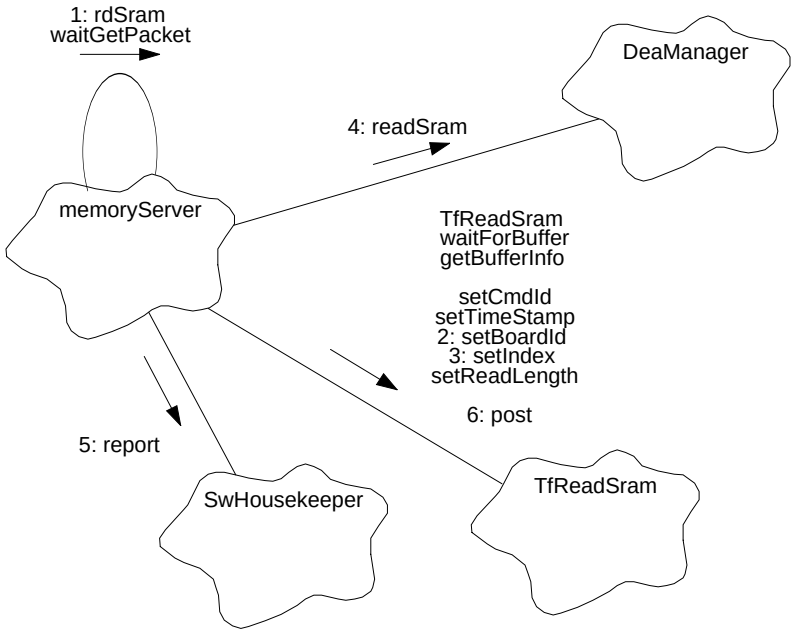
1. A request to read FEP data follows about the same scenario as in read BEP data. `goTaskEntry`() will call `rdFep`() to obtain a packet, will establish load points, and will install the specific header information.
2. The form's `setFepId`() is added to the heading to identify the intended FEP.
3. Provided with a target FEP identity, the address to be read from, a buffer address to load the data into and the amount to be loaded; the **`FepManager::`**`readMemory`() will be used to fill the buffer. It handles reads to I_cache, D_Cache or bulk memory.
4. Should the **`FepManager`** encounter a difficulty in delivering the requested data, it will return a Boolean value indicating the futility of continuing. `rdFep`() will use **`SwHouskeeper::`**`report`() to record the incident, will `release`() the packet buffer and return. Upon leaving scope, the packet instance is eliminated.
5. The forms' `post`() is used to instigate delivery of the packet. `rdFep`() determines when the request has been completed. It will continue to obtain a packet, load the header, and pass its buffer address and other arguments to `readMemory`() until the request has been completed. When the request is completed, control is returned to the main process where the semaphore is made available to the next request using **`sema-phore::`**`release`() (not shown).

27.4.4.3 Read from DEA Sequencer RAM

The DEA manager will support requests to deliver a copy of SRAM memory from one of the DEA CCD boards. The BEP will provide the packets needed to fulfill the request and will designate the packet storage buffer address. It will specify which DEA CCD board

memory is to be read, where to begin and how much may be stored in the buffer. Figure 128 illustrates this shared activity.

FIGURE 128. Process a read SRAM request

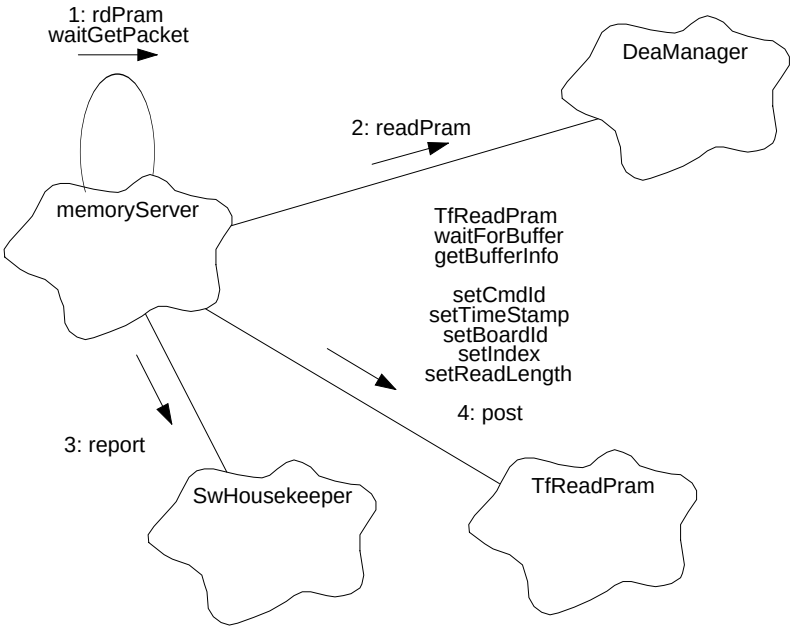


1. When a read from SRAM is requested, `goTaskEvent`() will pass responsibility to `rdSram`(). Again, a similar scenario is followed. A packet is obtained, the header information loaded, and the responsibility for loading the data is passed to the **`DeaManager`**.
2. The calls to obtain a packet and to load the header have the same name as previous read memory requests. The DEA unit identifier is installed by the forms' `setBoardId`().
3. The starting location of the data to be read into the packet buffer is installed using the forms' `setIndex`().
4. The **`DeaManager::`**readSram() is provided with the proper request to fill the buffer provided, and returns when the data has been installed.
5. Should the **`DeaManager`** encounter a difficulty in delivering the requested data, it will return an indication of the futility of continuing. `rdSram`() will use **`SwHousekeeper::`**`report`() to record the failure, will `release`() the packet buffer and return.
6. The forms' `post`() initiates delivery of the packet. `rdSram`() determines when the request has been completed. It will continue to obtain a packet, load the header, and pass its buffer address and other parameters to `readSram`() until the request has been complete. When the request is completed, control is returned to the main process where the semaphore is made available to the next request using **`semaphore::`**`release`() (not shown).

27.4.4.4 Read DEA Program RAM

The DEA manager will support requests to deliver a copy of PRAM memory from one of the DEA CCD boards. The memory service will provide the packets needed to fulfill the request and will designate the packet location to begin loading data. It will specify which DEA CCD board memory is to be read, where to begin and how much may be stored in the buffer. Figure 129 illustrates these actions.

FIGURE 129. Process a read PRAM request.



1. When a read from PRAM is requested, `rdPram`() is called. The scenario, again is similar; the calls to obtain a packet, and to load the header have the same names as in prior read requests.
2. The **`DeaManager::`**`readPram`() is called with appropriate arguments to have the provided packet buffer loaded.
3. If the **`DeaManager`** can not fulfill the request to deliver the data, it will return an indication of the failure. `rdPram`() will use **`SwHouskeeper::`**`report`() to record the condition, will `release`() the packet buffer and return.
4. The forms' `post`() is used to initiate delivery of the packet. `rdPram`() determines when the request has been completed. It will continue to obtain a packet, load the header, and pass its buffer address and other parameters to `readSram`() until the request has been fulfilled. When the request is completed, control is returned to the main process where the semaphore is made available to the next request using **`semaphore::`**`release`() (not shown).

27.4.5 Use 4:: Write memory of the BEP, FEP, PRAM, and SRAM

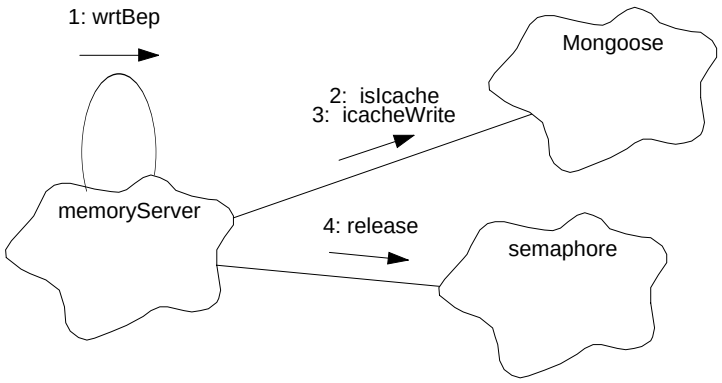
As with the read memory scenario, each of the write requests will: verify arguments, obtain the control semaphore, and notify the main process thread. This task, `goTaskEntry`(), passes control to the designated function. The request may be handled directly, or may be handed to the respective interfacing manager which assists by fulfilling the request. The size of each write is restricted by the assumption that it could have been contained within a 128 word command packet. However; there is no restriction to the number of sequential writes commanded, nor to the client requesting the service. The write requests do not generate telemetry packets initiated by the MemoryServer.

Section 27.4.2 describes the client request of a service by selecting the MemoryServer public procedure which specifies the service desired, delivering the location to be targeted, and the relevant data needed to fulfill the task. The public function has stored the data to be copied into a private buffer.

27.4.5.1 Write data to Back End Processor memory

Figure 130 illustrates the process of fulfillment by describing the steps needed to overwrite a section of the Back End Processors' memory.

FIGURE 130. Process a write BEP request

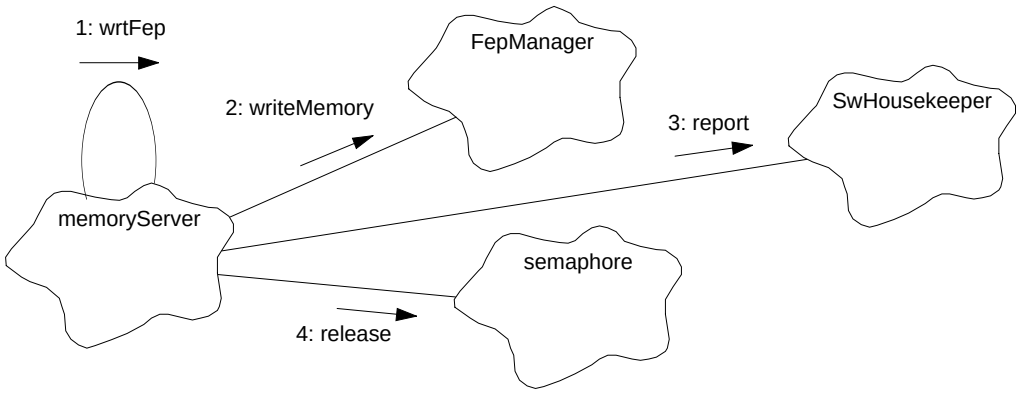


1. Upon being alerted by `notify`(), sent by the client through the public functions, the main process thread, `goTaskEvent`(), handles the request by directing the BEP writing service function, `wrtBep`(), to process the request.
2. The `wrtBep`() function uses `isIcache`() to test for an I_Cache write request so it may invoke the appropriate copying method.
3. `wrtBep`() will call `icachewrite`() to write data in to the I cache or else it will overwrite the memory directly itself.
4. With the request completed, control is returned to the main process, `goTaskEntry`() which will use **`semaphore::release`**() to free the semaphore and will return to wait for another request.

27.4.5.2 Write data to Front End Processor memory

Assistance from the FEP manager is required to copy data into the memory of one of the FEPs. The FEP manager will provide the copying service, whether to I_Cache, D_cache or bulk memory. The memory service will provide the address of the data and specify the location to be targeted. This is illustrated in Figure 131

FIGURE 131. Process a write FEP request



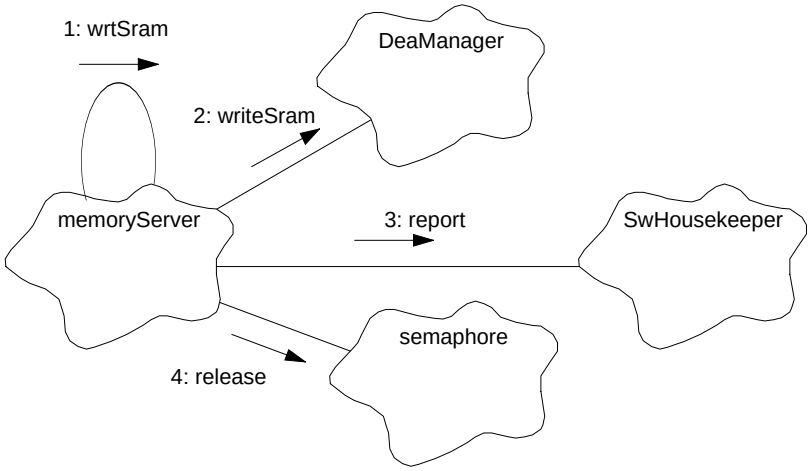
1. A request to read FEP data follows about the same scenario as read BEP, above. `goTaskEntry`() calls `wrtFep`() to manage the data copy to the FEP memory.
2. Provided with a target FEP identity, the buffer address that data is to be read from, the destination address that data is to be copied into and the amount to be copied; the `wrtFep`() will use **`FepManager::`**`writeMemory`() to fulfill the request.
3. Should the **`FepManager`** encounter a difficulty in delivering the requested data, it will return a Boolean value indicating a failure to complete its mission. `wrtFep`() will use **`SwHouskeeper::`**`report`() to record the incident and return.
4. `goTaskEntry`() will `release`() the semaphore and then wait for the next event notification to arrive.

27.4.5.3 Write data to DEA Sequencer RAM

The DEA manager will support requests to copy data into SRAM of one of the DEA CCD boards. The memory service will designate the buffer location from which to begin copying data. It will specify which DEA CCD board memory, data is to be written, where to

begin and how much of SRAM is to be over-written. Figure 132 illustrates this scenario.

FIGURE 132. Process a write SRAM request



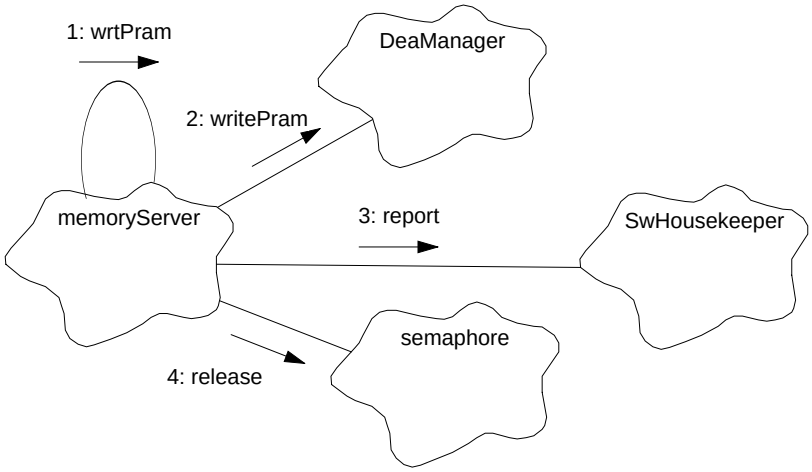
1. When a write to SRAM is requested, `goTaskEntry`() calls `wrtSram`() to manage the data copy to the DEA SRAM.
2. The **`DeaManager::`**`writeSram`() is called with appropriate arguments to have the designated DEA CCD board SRAM location loaded with the supplied data.
3. If the **`DeaManager`** can not fulfill the request to over-write the memory, it will return an indication of the failure. `wrtSram`() will use **`SwHouskeeper::`**`report`() to record the condition and return.
4. `goTaskEntry`() will use **`semaphore::`**`release`() to free the semaphore and then will wait for the next event notification to arrive.

27.4.5.4 Write data to DEA Program RAM

The DEA manager will support requests to copy data into PRAM of one of the DEA CCD boards. The memory service will designate the buffer location from which to begin copy-

ing data. It will specify which DEA CCD board memory is to be written to, where to begin and how much of PRAM is to be over-written. Figure 133 illustrates this scenario.

FIGURE 133. Process a write PRAM request



1. When a write to PRAM is requested, `goTaskEntry`() calls `wrtPram`() to manage the data copy to the DEA PRAM.
2. The **`DeaManager::`**`writePram`() is called with appropriate arguments to have the designated DEA CCD board PRAM location loaded with the supplied data.
3. If the **`DeaManager`** can not fulfill the request to over-write the memory, it will return an indication of the failure. `wrtPram`() will use **`SwHouskeeper::`**`report`() to record the condition and return.
4. `goTaskEntry`() will use **`semaphore::`**`release`() to free the semaphore and then wait for the next event.

27.4.6 Use 5:: Execute appropriate BEP or FEP memory

The execute functions public interface checks the input, screening out attempts to execute in D_Cache. It obtains the control semaphore, stores the arguments and the target location, then notifies the main process which will invoke the private method used to fulfill the request.

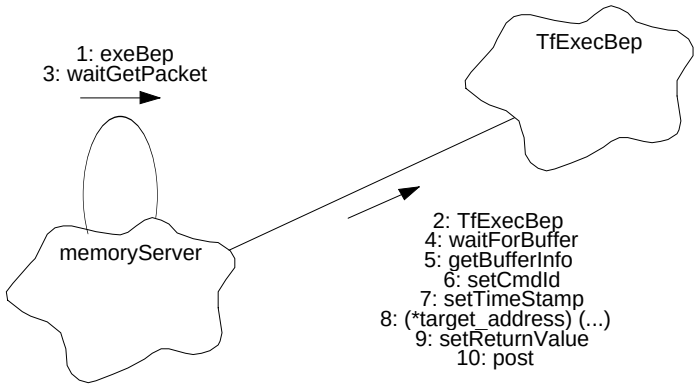
Note: the routine servicing this request will initiate execution of the target code as a sub-routine. Either the MemoryServer function or its surrogate will become the executing thread; unless the process it activates is initiated as an independent thread which allows its initiator to return. Either way, if the execution time approaches the watchdog duration, the executing procedure must be able to handle the `taskMonitor`() interrogations. Another approach would be to have the task `touch`() the watchdog directly.

27.4.6.1 Execute code in Back End Processor memory

The public execute BEP function provides for (up to) twenty arguments with which to begin execution at the I_Cache or bulk memory address provided. When the executing

process returns, the private fulfilling function, will include a one word (32bit) result in its telemetry buffer. Refer to Figure 134.

FIGURE 134. Process an execute BEP request



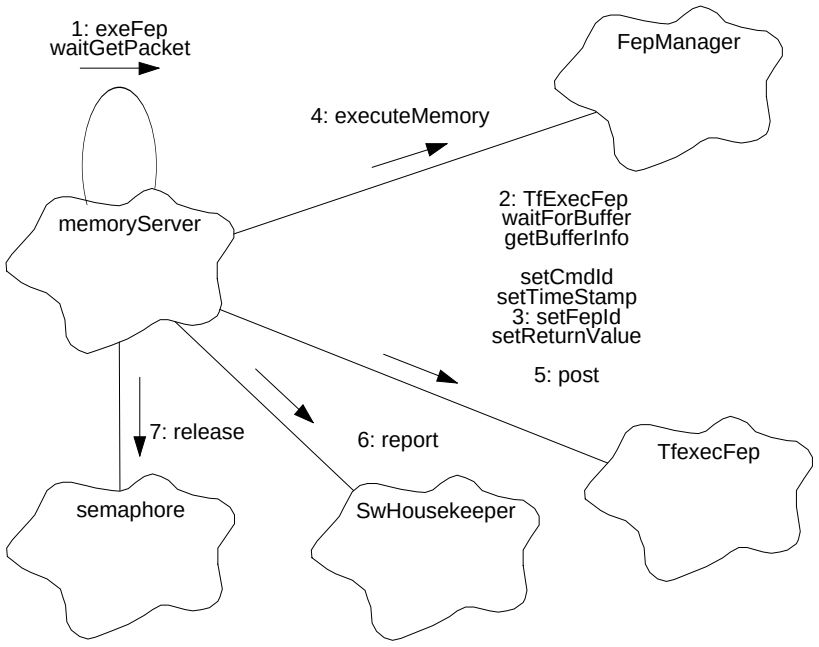
1. Upon being alerted by `notify`(), sent by the client through the public function, the main process, `goTaskEvent`(), handles the request by directing the BEP execute service function, `exeBep`(), to process the request.
2. It uses the form **`TfExecBep`** constructor to obtain a telemetry packet instance.
3. It calls the MemoryServer function `waitGetPacket`() to supply a packet buffer.
4. The subroutine checks whether the monitor has requested a response and uses `waitForBuffer`() which will linger for a stated period, before returning having succeeded or cycling to try again.
5. `exeBep`() uses the forms' `getBufferInfo`() to determine the location of the packet buffer and its length.
6. Then it uses the forms' `setCmdId`() to install the command identifier into the packet buffer.
7. The forms' `setTimeStamp`() is used to provide a timing reference.
8. `exeBep`() will call the memory location as an executable function `(*`*`target_address`*`)(...)` delivering the maximum number of arguments. The function, will only use the number it expects.
9. After the targeted process returns, `exeBep`() will call to the forms' `setReturnValue`() to have data returned by the function inserted into the buffer.
10. A call to the forms' `post`() will forward the packet for delivery.
11. When the request is completed, control is returned to the main process where the semaphore is made available to the next request using **`semaphore::`**`release`() (not shown). `goTaskEntry`() will then wait for the next event.

27.4.6.2 Execute code in Front End Processor memory

The scenario for executing in FEP is similar to that for BEP. This request, however, uses the assistance of the FEP manager to take the arguments and initiate the execution. The public execute FEP function acquires the arguments to begins execution at the I_Cache or

bulk memory address provided. The private fulfilling function, will include a one word (32bit) result in its telemetry buffer. Figure 135 illustrates this scenario.

FIGURE 135. Process an execute FEP request



1. Upon being alerted by `notify`(), sent by the client through the public function, the main process, `goTaskEvent`(), handles the request by directing the FEP execute service function, `exeFep`(), to process the request.
2. It uses the form **`TfExecFep`** constructor to obtain a telemetry packet instance, then, as with `exeBep`(), it calls the MemoryServer function `waitGetPacket`() to supply a packet buffer. The subroutine checks the monitor, as described in Section 27.4.3 , and uses `waitGetPacket`() which will linger for a stated period, before returning having succeeded or cycling to try again. Having obtained one, `exeFep`() uses the forms' `getBufferInfo`() to determine the location of the packet buffer and its length.
3. Then it uses the forms' `setCmdId`() to install the command identifier, `setTimeStamp`() to provide a timing reference, and `setFepId`() to identify the targeted FEP.
4. `exeFep`() will call the **`fepManager::executeMemory`**() specifying which FEP's memory to target, the memory location to begin execution, and the buffer location of the arguments to be used. When the result is returned, `exeFep`() will call the forms' `setReturnValue`() to have data word returned by the function inserted into the buffer.
5. A call to the forms' `post`() will forward the packet for delivery.
6. If the **`DeaManager`** can not fulfill the request to over-write the memory, it will return an indication of the failure. exeFep() will use **`SwHouskeeper::report`**() to record the condition and return.
7. `goTaskEntry`() will use **`semaphore::release`**() to free the semaphore and then will wait for the next event to arrive.

27.4.6.3 Execute code in memory - Caveat

Note: that the routine servicing this request will initiate execution of the target code as a subroutine. It will become the executing thread; unless the process it activates is initiated as an independent thread which allows its initiator to return. Either way, if the execution time approaches the watchdog duration, the executing procedure must be able to handle the `taskMonitor`() interrogations. Another approach might be to `touch`() the watchdog directly.

27.4.7 Use 6: Read and telemetry configuration information

This feature is implemented using the Read Back End Memory feature (see Section 27.4.4). In addition to specifying the address and length of the region of memory to telemeter, the caller passes the telemetry format tag to use with the telemetered memory. This allows the ground to easily identify the region of memory, and structure of the information being sent, without resorting to looking up the address of the region being dumped.

27.5 Class MemoryServer

Documentation:

The **`MemoryServer`** is responsible for accomplishing direct reads, writes, and execution of memory requested by explicit commands. The operations use a semaphore in performing their functions which guarantees serial execution of the commands. Requests received while an operation is in progress, are rejected and the public function returns "CMDRESULT_BUSY".

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Public Interface:

Operations:
```
MemoryServer()
executeBep()
executeFep()
goTaskEntry()
readBep()
readFep()
readPram()
readSram()
writeBep()
writeFep()
writePram()
writeSram()
```

Private Interface:

Has-A Relationships:

The *`rqst...`* variables will contain the information which is being passed from the commanding function to the task servicing the request.

`const unsigned*` *`rqstAddrR`***`::`** Target Address provided by the current request. This is the BEP or FEP address from which to begin reading.

`unsigned*` *`rqstAddrW`***`::`** Target Address provided by the current request. This is the BEP or FEP address to which to begin writing.

unsigned(* *rqstExeAddr* **)(...)** Address provided by the request. This is the BEP or FEP address at which to begin executing code.

unsigned *rqstIndex*:: Requested target index. This is the DEA index, location, from which to begin reading or to which to begin writing

unsigned *rqstCnt*:: Requested word or argument count. This is the number of items to be read or to be written; or the number of arguments being provided for execution of code.

FepId *rqstFepId*:: Requested hardware identifier. It is used to identify FEP to which the request is directed.

DeaCcdBdId *rqstDeaId*:: Requested hardware identifier. It is used to identify FEP to which the request is directed.

unsigned *rqstCmdId*:: Requests' command identification number.

unsigned *rqstMemBuf[]*:: The memory buffer which holds the data which will be used in fulfilling write or execute commands directed to the BEP or FEP.

DeaPramWord *rqstPramBuf[]*:: The memory buffer for use by write PRAM commands.

DeaSramWord *rqstSramBuf*[]:: The memory buffer for use by write SRAM commands.

void (* *memFuncPtr* **)()::** Denotes the MemoryServer service routine requested; rdBep(), wrtSram(), exeFep(), etc.

Semaphore *lock*:: Specifies the MemoryServer semaphore handle. The single semaphore insures serial processing of requests.

TlmFormatTag *rqstRdBepTag*:: This is the telemetry format tag to use when performing the current Back End Read Memory request.

Operations:

```
crossIBound()
exeBep()
exeFep()
getSemaphore()
rdBep()
rdFep()
rdPram()
rdSram()
waitGetPacket()
wrtBep()
wrtFep()
wrtPram()
wrtSram()
```

Concurrency: Active

Persistence: Persistent

27.5.1 MemoryServer()

Private member of: **`MemoryServer`**

Arguments:

`unsigned` *`taskId`*
`unsigned` *`lockId`*

Documentation:

This constructor initializes the **`MemoryServer`** instance. *`taskId`* is this task's identifier and *`lockId`* is the semaphore used by this task. `MemoryServer` initializes the global variables. Refer to the private interface listing in section 27.5 "Class MemoryServer".

Concurrency: Sequential

27.5.2 crossIBound()

Private member of: **`MemoryServer`**

Return Class: **`CmdResult`**

Arguments:

`const unsigned*` *`addr`*
`unsigned` *`wordCnt`*

Documentation:

`crossIBound` performs the checks to validate legal BEP & FEP addressing.

Semantics:

`crossIBound` checks the requested read/write range to insure that it does not cross into, or out of the I_cache boundary. The request must be totally within a memory area (I_Cache being between D_Cache and bulk memory). This function also checks that the address is on a word boundary.

Concurrency: Guarded

27.5.3 exeBep()

Private member of: **MemoryServer**

Return Class: **void**

Documentation:

exeBep initiates execution at the address provided with the arguments provided and returns the resultant value.

Semantics:

exeBep will obtain a **TfExecBep** telemetry packet instance, and will use waitGetPacket to obtain a telemetry packet. It will use **TfExecBep::**getReadBuffer to obtain the packet storage location and its length. exeFep stores: the command identity from *rqstCmdId* into the buffer using **TfCmdResponse::**setCmdId, the BEP tick count from **SystemClock::**currentTime into the buffer using **TfCmdResponse::**setTimestamp. It then references the specified number of arguments (stored in *rqstMemBuf*), and begins execution at the memory location provided in *rqstExeAddr*. The code is executed as a subroutine of the memory server thread. When execution completes, it will install the return value into the telemetry buffer using **TfExecBep::**setReturnValue, then hand the packet off to the telemetry manager using **TlmForms::**post.

Note: if the execution time approaches the watchdog interval, provision must be made by the user, to have the executing process handle the taskMonitor interrogations.

Concurrency: Guarded

27.5.4 executeBep()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

```
unsigned cmdIdent
unsigned(*execAddr)(...)
const unsigned* argBuff
unsigned argCount
```

Documentation:

executeBep instructs the **MemoryServer** to store the accompanying arguments, and to initiate execution at the address provided.

Preconditions:

The address to execute must not be within D_Cache.

Semantics:

executeBep verifies that the address is not in D_Cache, and that the number of arguments is within bounds, else softwareHousekeeping is called and the status CMDRESULT_BAD_ARGUMENT is returned. executeBep attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the code address to begin execution; *execAddr* to *rqstExecAddr*, the number of arguments to copy; *argCount* to *rqstCnt*, and the arguments to be provided are copied from *argBuff* into *rqstMemBuf*. The address of the fulfilling function exeBep is stored in *memFunsPtr*. The **MemoryServer** event is set to notify goTaskEntry(), and the status CMDRESULT_OK is returned.

Postconditions:

The arguments and the location to be executed have been made available and the thread notified.

Concurrency: Synchronous

27.5.5 exeFep()

Private member of: **MemoryServer**

Return Class: **void**

Documentation:

exeFep initiates execution by the Front End Process Manager, **fepManager**, on the designated FEP at the memory address provided, and with the arguments provided. Then it telemeters the returned result.

Semantics:

exeFep will obtain a **TfExecFep** telemetry packet instance, and will use waitGetPacket to obtain a telemetry packet. It will use **TfExecFep::**getReadBuffer to obtain the packet storage location and its length. exeFep stores: the command identity from *rqstCmdId* into the buffer using **TfCmdResponse::**setCmdId; the BEP tick count from **SystemClock::**currentTime into the buffer using **TfCmdResponse::**setTimestamp; the FEP identity into the buffer using **TfReadFep::**setFepIdentity. It then provides a pointer to the arguments (stored in *rqstMemBuf*), and directs the FEP manager to execute the code at the memory location provided in *rqstExeAddr* by calling **fepManager::**executeMemory. If the **fepManager** could not fulfill the request, notification is made by sending a **SwHouskeeper::**report; the packet will be released back to the pool from **TlmForms::**~TlmForm when control passes from the scope of exeFep. With this request completed, exeFep will install the return value into the telemetry buffer using **TfExecFep::**setReturnValue, then it will hand the packet off to the telemetry manager using **TlmForms::**post.

Note: if the execution time approaches the watchdog interval, the provision must be made by the user, to have the executing process handle the taskMonitor interrogations.

Concurrency: Guarded

27.5.6 executeFep()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

```
unsigned cmdIdent
FepId identFep
unsigned(*execAddr)(...)
const unsigned* argBuff
unsigned argCount
```

Documentation:

executeFep instructs the **MemoryServer** to take the accompanying arguments, and initiate execution on the designated FEP at the address provided.

Preconditions:

The address to execute must not be within D_Cache.

Semantics:

executeBep verifies that the address is not in D_Cache, that the target address is on a word boundary, and the number of arguments is within bounds, else SwHousekeeper is called and the status CMDRESULT_BAD_ARGUMENT is returned. executeBep attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the status CMDRESULT_BUSY is returned

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the code address to begin execution; *execAddr* to *rqstExecAddr*, the number of arguments to copy; *argCount* to *rqstCnt*, and the arguments to be provided are copied from *argBuff* into *rqstMemBuf*. The address of the fulfilling function exeFep is stored in *memFunsPtr*. The **MemoryServer** event is set to notify goTaskEntry(), and the status CMDRESULT_OK is returned.

Postconditions:

The arguments and the location to be executed have been made available and the thread notified.

Concurrency: Synchronous

27.5.7 goTaskEntry()

Public member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`goTaskEntry` is initialized as the **`MemoryServer`** thread, and leaves the process waiting for its events. When alerted, it invokes the appropriate fulfilling function or `taskMonitor` response. On the functions return, it releases the semaphore and returns to waiting for an event.

Semantics:

The process waits FOREVER for its event, EV_REQUEST_SERVICE, or for the `taskMonitor` query, EV_TASKQUERY; to which it replies using **`taskMonitor::`**`respond`. When its event is received, it consumes the event, verifies and then invokes the appropriate function, which was designated during execution of the public function. On the subroutines' return; `goTaskEntry` releases the semaphore, invalidates the function address, and cycles to wait for an event.

Postconditions:

The process is waiting FOREVER for one of its events or for the `taskMonitor` query request.

Concurrency: Sequential

27.5.8 getSemaphore()

Private member of: **`MemoryServer`**

Return Class: **`CmdResult`**

Documentation:

`getSemaphore` attempts to obtain the **`MemoryServer`** semaphore which is required before executing the requested service.

Semantics:

`getSemaphore` uses the **`Semaphore::`**`request` function when attempting to obtain the semaphore; succeeding it returns CMDRESULT_OK, else it returns CMDRESULT_BUSY.

Concurrency: Guarded

27.5.9 readBep()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

unsigned *cmdIdent*
Const unsigned* *addr*
unsigned *readCount*
TlmFormatTag *tag*

Documentation:

readBep() posts a request to the **MemoryServer** main thread goTaskEntry() to initiate a memory read from the Back End Processor memory at the address provided. The service will use rdBep() to deliver the number of words requested.

Semantics:

The target area specifications are verified using crossIBound(), else the status DATA_ERROR is returned. The **MemoryServer** semaphore must be acquired or the status CMDRESULT_BUSY is returned. Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the address to begin reading from; *addr* to *rqstAddrR*, and the number of 32 bit words to read and deliver; *readCount* to *rqstCnt*. The address of the fulfilling function rdBep() is stored in *memFuncPtr*. The telemetry format tag to use when sending the region, *tag*, is stored in *rqstRdBepTag*. The **MemoryServer** event is set to notify the autonomous task goTaskEntry(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The beginning location and the amount of data to be delivered have been made available in global variables and the thread notified. The total amount of memory to be read will be delivered in one or more packets.

Concurrency: Synchronous

27.5.10 readFep()

Public member of: **MemoryServer**

Return Class:

CmdResult

Arguments:

unsigned *cmdIdent*
FepId *identFep*
const unsigned* *addr*
unsigned *readCount*

Documentation:

`readFep` instructs the MemoryServer to initiate a memory read of a number of words by the specified Front End Processor at the address provided.

Preconditions:

Memory to be read must not cross the D_Cache / I_Cache boundary, nor the I_Cache / Bulk boundary.

Semantics:

The target area specification are verified using `crossIbound`. Any error conditions are passed to softwareHousekeeping, and the status CMDRESULT_BAD_ARGUMENT is returned.

The **`memoryServer`** semaphore must be acquired or the status CMDRESULT_BUSY is returned. Having acquired the semaphore, the command data are stored: the command identifier; *`cmdIdent`* to *`rqstCmdId`*, the FEP from which the memory is to be read; *`identFep`* to *`rqstFepId`*, the address to begin reading from; *`addr`* to *`rqstAddrR`*, and the number of 32 bit words to read and deliver; *`readCount`* to *`rqstCnt`*. The address of the fulfilling function `rdFep`() is stored in *`memFuncPtr`*. The **`MemoryServer`** event is set to notify the autonomous task `goTaskEntry`(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The beginning location and the amount of data to be delivered have been made available and the task notified.

Concurrency: Synchronous

27.5.11 readPram()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

unsigned *cmdIdent*
DeaCcdBdId *identPram*
unsigned *index*
unsigned *readCount*

Documentation:

`readPram` instructs the **MemoryServer** to initiate a memory read of a specified number of words from the identified DEA board PRAM by the DEA manager, `deaManager`.

Semantics:

The validity of the arguments having been checked beforehand, `readPram` attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the return status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the DEA from which the memory is to be read; *identPram* to *rqstDeaId*, the index to begin reading from; *index* to *rqstIndex*, and the number of PRAM words to read and deliver; *readCount* to *rqstCnt*. The address of the fulfilling function `rdPram`() is stored in *memFuncPtr*. The **MemoryServer** event is set to notify the autonomous task `goTaskEntry`(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The beginning location and the amount of data to be delivered have been made available and the thread notified.

Concurrency: Synchronous

27.5.12 readSram()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

unsigned *cmdIdent*
DeaCcdBdId *identSram*
unsigned *index*
unsigned *readCount*

Documentation:

`readSram` instructs the MemoryServer to initiate a memory read of a specified number of words from the identified DEA board SRAM by the DEA manager, `deaManager`.

Semantics:

The validity of the arguments having been checked beforehand, `readSram` attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the return status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the DEA from which the memory is to be read; *identSram* to *rqstDeaId*, the index to begin reading from; *index* to *rqstIndex*, and the number of SRAM words to read and deliver; *readCount* to *rqstCnt*. The address of the fulfilling function `rdSram`() is stored in *memFuncPtr*. The **MemoryServer** event is set to notify the autonomous task `goTaskEntry`(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The beginning location and the amount of data to be delivered have been made available and the thread has been notified.

Concurrency: Synchronous

27.5.13 rdBep()

Private member of: **MemoryServer**

Return Class: **void**

Documentation:

`rdBep` reads BEP memory and loads it into telemetry buffers.

Semantics:

`rdBep` determines if the address from which to read data is in I_cache using **`mongoose::`**`isIcache`. It then enters a loop, obtains a **`TfReadBep`** telemetry packet instance, and uses `waitGetPacket` which returns when the packet buffer is available. `rdBep` uses `getReadBuffer` to obtain the packet storage location and its length. `rdBep` stores: the command identity from *`rqstCmdId`* into the buffer using **`TfCmdResponse::`**`setCmdId`, the BEP tick count from **`SystemClock::`**`currentTime` into the buffer using **`TfCmdResponse::`**`setTimestamp`, copies the target address which begins this read, and the length of data to be read, into the buffer using **`TfReadBep::`**`setMemAddr` and **`TfReadBep::`**`setReadLength`, respectively. It reads the memory, writing it into the buffer directly or by using the I_Cache read memory utility **`Mongoose::`**`icacheRead`. `rdBep` uses **`TlmForms::`**`post` to hand the packet off to the telemetry manager. Having adjusted the address for the amount of memory copied, it decrements that amount from the requested length. This cycle continues until the request has been fulfilled.

Concurrency: Guarded

27.5.14 rdFep()

Private member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`rdFep` oversees reading of FEP memory by the FEP manager and provides and forwards telemetry buffers.

Semantics:

`rdFep` enters a loop, obtains a **`TfReadFep`** telemetry packet instance, and uses `waitGetPacket` to request a telemetry packet buffer, returning when one is available. `rdFep` uses **`TfReadFep::`**getReadBuffer to obtain the packet storage location and its length. `rdFep` stores: the command identity from *`rqstCmdId`* into the buffer using **`TfCmdResponse::`**`setCmdId`; the BEP tick count from **`SystemClock::`**`currentTime` into the buffer using **`TfCmdResponse::`**`setTimestamp`; the FEP identity into the buffer using **`TfReadFep::`**`setFepIdentity`; and copies the target address which begins this read, and the length of data to be read, into the buffer using **`TfReadFep::`**`setMemAddr` and **`TfReadFep::`**`setReadLength`, respectively. It determines the length of data to be loaded into the packet buffer and calls the Front End Processor Manager requesting that memory be copied from the designated FEPs' specified address to the telemetry buffer address using **`fepManager::`**`readMemory`. When the manager returns, if it could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`; the packet will be released back to the pool from **`TlmForms::`**`~TlmForm` when control passes from control of the loop in which it was invoked. With this request completed, `rdFep` will hand the packet to the telemetry manager using **`TlmForms::`**`post`, decrement the length of memory to be read, and increment the address. This cycle continues until all of the request has been fulfilled.

Concurrency: Guarded

27.5.15 rdPram()

Private member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`rdPram` initiates a memory read by the Detector Electronics Assembly Manager, **`deaManager`**, from the designated DEA at the index provided,

and telemeters the returned result.

Semantics:

`rdPram` begins a loop, obtains a **`TfReadPram`** telemetry packet instance, passing *`rqstRdBepTag`* as the tag to use for packets formatted by the instance, and uses `waitGetPacket` to request a telemetry packet buffer, returning when one is available. `rdPram` uses **`TfReadPram::`**`getReadBuffer` to obtain the packet storage location and its length. `rdPram` stores: the command identity from *`rqstCmdId`* into the buffer using **`TfCmdResponse::`**`setCmdId`; the BEP tick count from **`SystemClock::`**`currentTime` into the buffer using **`TfCmd-Response::`**`setTimestamp`; and the DEA identity into the buffer using **`TfReadPram::`**`setBoardId`. It determines the length of data to be loaded into the packet buffer then copies the target index at which this read begins, and the length of data to be read, into the buffer using **`TfRead-Pram::`**`setIndex` and **`TfReadPram::`**`setReadCount`, respectively. `rdPram` calls the Detector Electronics Assembly Manager requesting that memory be copied from the designated DEA boards' specified index to the telemetry buffer address using **`deaManager::`**`readPram`. When the manager returns, if it could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`; the packet buffer will be released back to the pool from **`TlmForms::`**`~TlmForm` when control passes from the scope of the loop in which it was invoked. With this request completed, `rdPram` will hand the packet to the telemetry manager using **`Tlm-Forms::`**`post`, decrement the count of PRAM words to be read, and increment the index from which to read. This cycle continues until the request has been fulfilled.

Concurrency: Guarded

27.5.16 rdSram()

Private member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`rdSram` initiates a memory read by the Detector Electronics Assembly

Manager, **deaManager**, from the designated DEA board at the index provided, and telemeters the returned result. rdSram provides the packet buffers and posts them when filled.

Semantics:

rdSram begins a loop, obtains a **TfReadSram** telemetry packet instance, and uses waitGetPacket to request a telemetry packet buffer, returning when one is available. rdSram uses **TfReadSram::**getReadBuffer to obtain the packet storage location and its length. rdSram stores: the command identity from *rqstCmdId* into the buffer using **TfCmdResponse::**setCmdId; the BEP tick count from **SystemClock::**currentTime into the buffer using **TfCmdResponse::**setTimestamp; and the DEA identity into the buffer using **TfReadSram::**setBoardId. It determines the length of data to be loaded into the packet buffer then copies the target index at which this read begins, and the length of data to be read, into the buffer using **TfReadSram::**setIndex and **TfReadSram::**setReadCount, respectively. rdSram calls the Detector Electronics Assembly Manager requesting that memory be copied from the designated DEA boards' specified index to the telemetry buffer address using **deaManager::**readSram. When the manager returns, if it could not fulfill the request, notification is made by sending a **SwHouskeeper::**report; the packet buffer will be released back to the pool from **TlmForms::**~TlmForm when control passes from the scope of the loop in which it was invoked. With this request completed, rdSram will hand the packet to the telemetry manager using **TlmForms::**post, decrement the count of SRAM words to be read, and increment the index from which to read. This cycle continues until the request has been fulfilled.

Concurrency: Guarded

27.5.17 waitGetPacket()

Private member of: **MemoryServer**

Return Class: **void**

Arguments:

TlmForm &*pkt*

Documentation:

waitGetReply waits for a packet to become available while insuring that

taskMonitor queries are acknowledged.

Semantics:

`waitGetPacket` begins with a loop that checks the `taskMonitor` EV_TASKQUERY event and responds if it is set. It then waits using **`TlmForm::`**`waitForBuffer` for a period specified by GET_PKT_TIME_OUT, for a packet buffer to become available. When one is available, it returns and `pkt` has a buffer. If the wait times out before a packet is available, it cycles, checking `taskMonitor` query and waiting again for a packet to become available. `waitGetPacket` does not return without a packet.

Postconditions:

A packet instance with generic header and memory buffer is available to be filled with the requested data.

Concurrency: Guarded

27.5.18 writeBep()

Public member of: **`MemoryServer`**

Return Class: **`CmdResult`**

Arguments:

```
unsigned cmdIdent
unsigned* addr
const unsigned* srcBuff
unsigned writeCount
```

Documentation:

`writeBep` instructs the **`MemoryServer`** to initiate a memory write of the associated data to the Back End Processor address provided.

Preconditions:

Memory to be written must be wholly within D_Cache or within I_Cache or within Bulk. It may not cross the I_Cache boundary.

Semantics:

The target area specifications are verified using `crossIBound`, else the status CMDRESULT_BAD_ARGUMENT is returned. `writeBep` confirms that the amount of data to be written will fit in the private buffer, else it returns `CMDRESULT_BAD_ARGUMENT.` It attempts to acquire the **`MemoryServer`** semaphore. If it fails to acquire, the return status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *`cmdIdent`* to *`rqstCmdId`*, the address at which to begin writing; *`addr`* to *`rqstAddrW`*, and the number of 32 bit words to write and deliver; *`writeCount`* to *`rqstCnt`*. The address of the fulfilling function `wrtBep`() is stored in *`memFuncPtr`*. The **`MemoryServer`** event is set to notify the autonomous task `goTaskEntry`(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The total data to be written and its location have been made available and the thread notified.

Concurrency: Synchronous

27.5.19 writeFep()

Public member of: **`MemoryServer`**

Return Class: **`CmdResult`**

Arguments:

```
unsigned cmdIdent
FepId identFep
unsigned* addr
const unsigned* srcBuff
unsigned writeCount
```

Documentation:

writeFep instructs the **MemoryServer** to initiate a memory write of the associated data into the specified Front End Processor memory at the address provided.

Memory to be written must be within D_Cache or within I_Cache or within Bulk. It must not cross the I_Cache boundary

Semantics:

The target area specifications are verified using crossIBound, and writeBep confirms that the amount of data to be written will fit in the private buffer, else it returns CMDRESULT_BAD_ARGUMENT. It attempts to acquire the MemoryServer semaphore. If it fails to acquire, the return status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the FEP who's memory is to be overwritten; *identFep* to *rqstFepId* the address at which to begin writing; *addr* to *rqstAddrW*, and the number of 32 bit words to write and deliver; *writeCount* to *rqstCnt*, and the data to be written is copied from *srcBuff* into *rqstMemBuf*. The address of the fulfilling function wrtFep() is stored in *memFuncPtr*. The **MemoryServer** event is set to notify goTaskEntry(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The total data to be written and its location have been made available and the thread notified.

Concurrency: Synchronous

27.5.20 writePram()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

unsigned *cmdIdent*
DeaCcdBdId *identPram*
unsigned *index*
const DeaPramWord* *srcBuff*
unsigned *writeCount*

Documentation:

writePram instructs the **MemoryServer** to initiate a memory write of the associated data to the identified DEA board PRAM by the DEA manager.

Semantics:

The length of data to be written is checked against the local buffer size. If the data exceeds the buffer size, softwareHousekeeping is called and the status CMDRESULT_BAD_ARGUMENT is returned. writeSram attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the DEA who's memory is to be overwritten; *identPram* to *rqstDeaId* the index at which to begin writing; *index* to *rqstIndex*, and the number of PRAM words to write and deliver; *writeCount* to *rqstCnt*, and the data to be written is copied from *srcBuff* into *rqstPramBuf*. The address of the fulfilling function wrtPram() is stored in *memFuncPtr*. The **MemoryServer** event is set to notify goTaskEntry(), and the status CMDRESULT_OK is returned.

Postconditions:

The total data to be written and its location have been made available and the thread notified.

Concurrency: Synchronous

27.5.21 writeSram()

Public member of: **MemoryServer**

Return Class: **CmdResult**

Arguments:

unsigned *cmdIdent*
DeaCcdBdId *identSram*
unsigned *index*
const DeaSramWord* *srcBuff*
unsigned *writeCount*

Documentation:

writeSram instructs the **MemoryServer** to initiate a memory write of the associated data to the identified DEA board SRAM by the DEA manager.

Semantics:

The length of data to be written is checked against the local buffer size. If the data exceeds the buffer size, softwareHousekeeping is called and the status CMDRESULT_BAD_ARGUMENT is returned. writeSram attempts to acquire the **MemoryServer** semaphore. If it fails to acquire, the status CMDRESULT_BUSY is returned.

Having acquired the semaphore, the command data are stored: the command identifier; *cmdIdent* to *rqstCmdId*, the DEA who's memory is to be overwritten; *identSram* to *rqstDeaId* the index at which to begin writing; *index* to *rqstIndex*, and the number of SRAM words to write and deliver; *writeCount* to *rqstCnt*, and the data to be written is copied from *srcBuff* into *rqstSramBuf*. The address of the fulfilling function wrtSram() is stored in *memFuncPtr*. The **MemoryServer** event is set to notify goTaskEntry(), and the delivered status CMDRESULT_OK is returned.

Postconditions:

The total data to be written and its location have been made available and the thread notified.

Concurrency: Synchronous

27.5.22 wrtBep()

Private member of: **MemoryServer**

Return Class: **void**

Documentation:

`wrtBep` writes data into BEP memory.

Semantics:

`wrtBep` determines if the address from which to read data is in I_Cache using **`mongoose::isIcache`**, writes the data provided (previously stored in *`rqstMemBuf`*) into memory beginning at the address specified in *`rqstAddrW`* either directly or by using the I_Cache read memory utility **`mongoose::icacheWrite`**. The maximum data length is limited to the maximum command packet data space. There is no telemetered response.

Concurrency: Guarded

27.5.23 wrtFep()

Private member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`wrtFep` over sees data write by the designated FEP into memory.

Semantics:

`wrtFep` calls the Front End Processor Manager requesting a write by the designated FEP of the referenced data (stored in *`rqstMemBuf`*) beginning at the specified address using **`fepManager::`**`writeMemory`. When the manager returns, if the FEP manager could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`. When the manager returns, if it could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`. Then this process returns.

Concurrency: Guarded

27.5.24 wrtPram()

Private member of: **`MemoryServer`**

Return Class: void

Documentation:

`wrtPram` oversees write of the provided data by the Detector Electronics Assembly Manager, `deaManager`, to the specified DEA subsection PRAM at the index provided.

Semantics:

`wrtPram` calls the Detector Electronics Assembly Manager requesting a write of PRAM words contained in *`rqstPramBuf`* to the designated DEA board PRAM index. When the manager returns, if it could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`. Then this process returns.

Concurrency: Guarded

27.5.25 wrtSram()

Private member of: **`MemoryServer`**

Return Class: **`void`**

Documentation:

`wrtSram` oversees write of the provided data by the Detector Electronics Assembly Manager, `deaManager`, to the specified DEA subsection SRAM at the index provided.

Semantics:

`wrtSram` calls the Detector Electronics Assembly Manager requesting a write of SRAM words contained in *`rqstSramBuf`* to the designated DEA board SRAM index. When the manager returns, if it could not fulfill the request, notification is made by sending a **`SwHouskeeper::`**`report`. Then this process returns.

Concurrency: Guarded

28.0 Software Housekeeper (36-53220 A)

28.1 Purpose

The Software Housekeeper provides the statistical repository for the various active tasks. Periodically it delivers the database to be telemetered, and begins a fresh accumulation of software data. This process is initiated during start-up and persists until CPU reset.

The frequency with which the housekeeper attempts to deliver packets is one per minute (patchable).

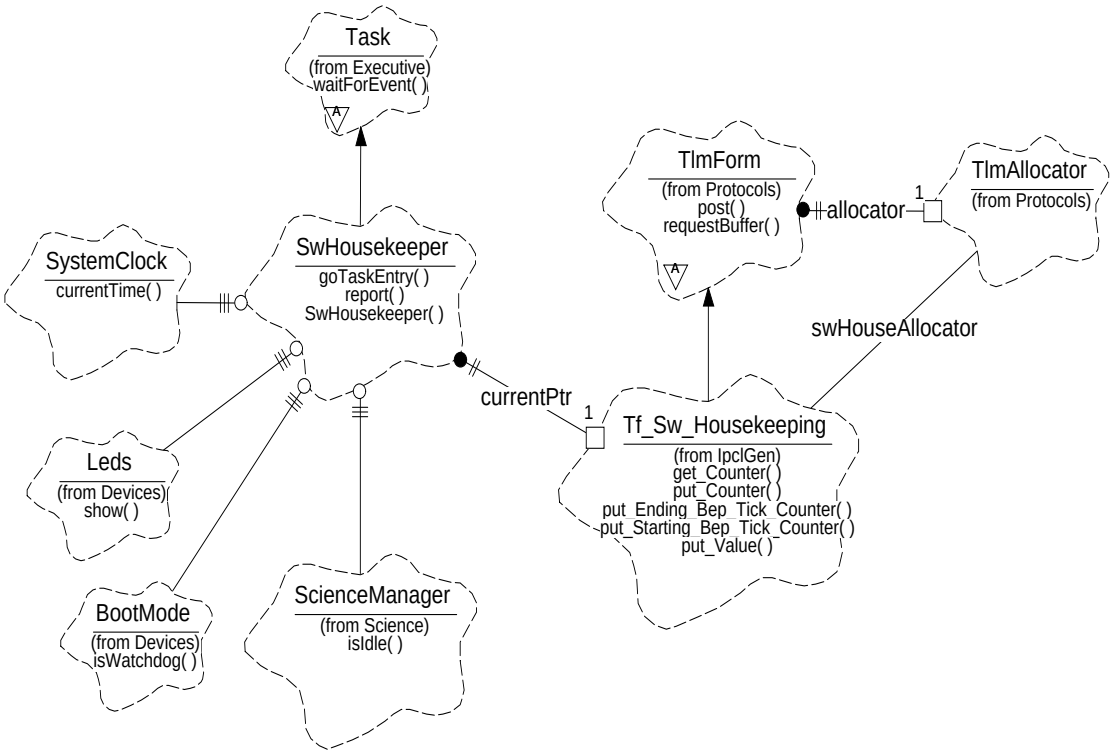
28.2 Uses

Use 1:: A method of acquiring reported statistical values
Use 2:: A system for periodically delivering the accumulated statistics.
Use 3:: Periodically indicate instrument state using software discrete telemetry (LEDs)

28.3 Organization

Figure 136 illustrates the relationship between the classes used by the Software Housekeeper.

FIGURE 136. Software Housekeeper Class Relationships



The Software Housekeeper uses the ***Executive***, ***Protocols***, ***Science*** and ***Devices*** class categories.

SwHousekeeper - This class is a subclass of ***Executive*****::*****Task***. It is responsible for accumulating and delivering software housekeeping statistics reported by other processes. It provides functions which clients use to report statistics (`report`), and includes a main task function (`goTaskEntry`).

Tf_Sw_Housekeeping - This class encapsulates the representation of a telemetry packet. It is a subclass of ***Protocols*****::*****TlmForm*** and is generated by the IP&CL code generator. The **SwHousekeeper** contains two instances of this class, which are used to format and post software housekeeping telemetry packet buffers. It provides functions which write the starting and ending integration times (`put_Starting_Bep_Tick_Counter`, `put_Ending_Bep_Tick_Counter`), read and write the current statistic counter (`get_Counter`, `put_Counter`), and set the value field associated with the statistic (`put_Value`).

SystemClock - This class provides the Back End Processor tick count (`currentTime`) which is included with the Software Housekeeping Database. It is provided by the ***Executive*** class category.

Leds - This class is provided by the ***Devices*** class category and is responsible for setting the current software discrete telemetry value (`show`). The **SwHousekeeper** class uses this class to indicate the current state of the instrument software.

BootMode - This class is provided by the ***Devices*** class category and is responsible for providing the cause of the most recent reset of the Back End Processor. The **SwHousekeeper** uses this class to determine if the Watchdog timer caused the most recent reset of the Back End (`isWatchdog`). This information is used to indicate the current state of the instrument in the software discrete telemetry.

ScienceManager - This class is provided by the ***Science*** class category and is responsible for managing science runs. The **SwHousekeeper** uses this class to determine whether or not a science run is active (`isIdle`) This information is used to indicate the current state of the instrument in the software discrete telemetry.

TlmAllocator - This class is provided by the ***Protocols*** class category and is responsible for managing pools of telemetry packet buffers. All software housekeeping telemetry packets are allocated from a single pool, managed by the *swHouseAllocator* instance of this class.

TaskMonitor - This class (not shown) is a subclass of ***Executive*****::*****Task***. It is responsible for insuring that active tasks are responsive, (not subverted by an SEU, etc.).

28.4 Scenarios

28.4.1 Operational Overview

The Software Housekeeper task has a quiescent period during which statistics are accumulated. When activated, it initiates an attempt to obtain a fresh database; failing, it continues accumulating into the current telemetry packet data array; succeeding, a fresh new array becomes available and the former packet is delivered to be telemetered. Since statistics are accumulated directly into telemetry buffers, they are subject to single-event upsets (SEUs). The expected SEU rate into the 1Mbyte telemetry buffer memory is from 1 to 100 hits per day, and the maximum size of a single telemetry buffer is 4092 bytes. A housekeeping data point may be corrupted about once every two or three days.

On each attempt, the housekeeper sets the software discrete telemetry bits (LEDs) to indicate the current state of the instrument (see Section 4.3.2).

These states include:

Science is idle

Science is running

Instrument was reset by the watchdog timer and Science is idle

Instrument was reset by the watchdog timer and Science is running

Each of these states use two different discrete telemetry codes. The housekeeper switches between each code on each iteration to indicate to the ground that the housekeeper is active (see the ACIS Software IP&CL, MIT 36-5302.0204, for the formal list of code assignments).

The software discrete bi-levels are periodically sampled by the spacecraft via the RCTU, and are included in the engineering portion of telemetry. The sample rate is TBD, but is expected to be once per Major Frame (i.e. about twice a minute). The final location of the bi-level signals within the telemetry frame, and their sample rate will be specified in the final version of the AXAF-I Instrument Program and Command List (IP&CL), provided by TRW (part # TBD).

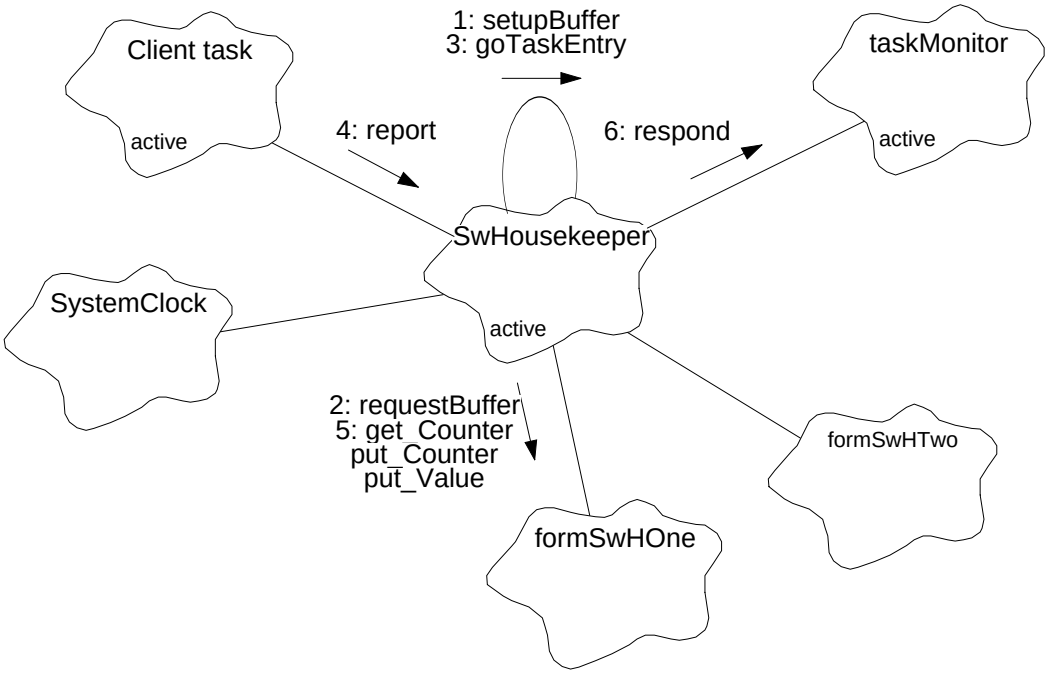
Each of the housekeeping statistics are comprised of two items; the number of times each entry was reported during the period, and an associated value. An enumerated list is used to specify the data items reported. The accumulating process will increment the count of occurrences in the first item of the two item data structure installed at the enumerated database location as each incident is deposited. The associated values' meaning and representation is entirely application dependent. It is stored as an unsigned integer.

Any process may be a client of the software housekeeper. It is the clients responsibility to deliver whichever statistical data are desired. The housekeeper does not solicit items to be included in the statistics.

28.4.2 Use 1: A method of acquiring reported statistical values

Figure 137 illustrates the operation of the Software Housekeeper data accumulation process.

FIGURE 137. Software Housekeeper Accumulation



1. During the system initialization, the constructor, `SwHousekeeper`(), passes a pointer to *`formSwHOne`* to **`SwHousekeeper::`**`setupBuffer`() to initialize the first packet buffer. Once the buffer is setup, it sets the form pointer, *`currentPtr`*, to point to *`formSwHOne`*. Once running, the housekeeper switches this pointer between *`formSwHOne`* and *`formSwHTwo`* to allow statistic accumulation while the other's telemetry packet buffer is being sent.
2. `setupBuffer`() uses *`formSwHOne`*.`requestBuffer`() to obtain the initial telemetry packet buffer, and initializes the contents of the buffer (see Section 28.4.3).
3. Once the system starts multi-tasking, the housekeeper's task function, `goTaskEntry`() is invoked. This function contains an infinite loop during which it waits in `intervalWait`() (not shown) while statistical data accumulates, then delivery of the housekeeping database packet is initiated.
4. The client task, any active software process, may deposit information whenever necessary, or desirable using **`SwHousekeeper::`**`report`().
5. The software housekeeper `report`() function disables interrupts by declaring an instance of **`IntrGuard`** (not shown). It then reads the current statistic value using *`currentPtr`*`->get_Counter`(), increments the returned value and writes it back

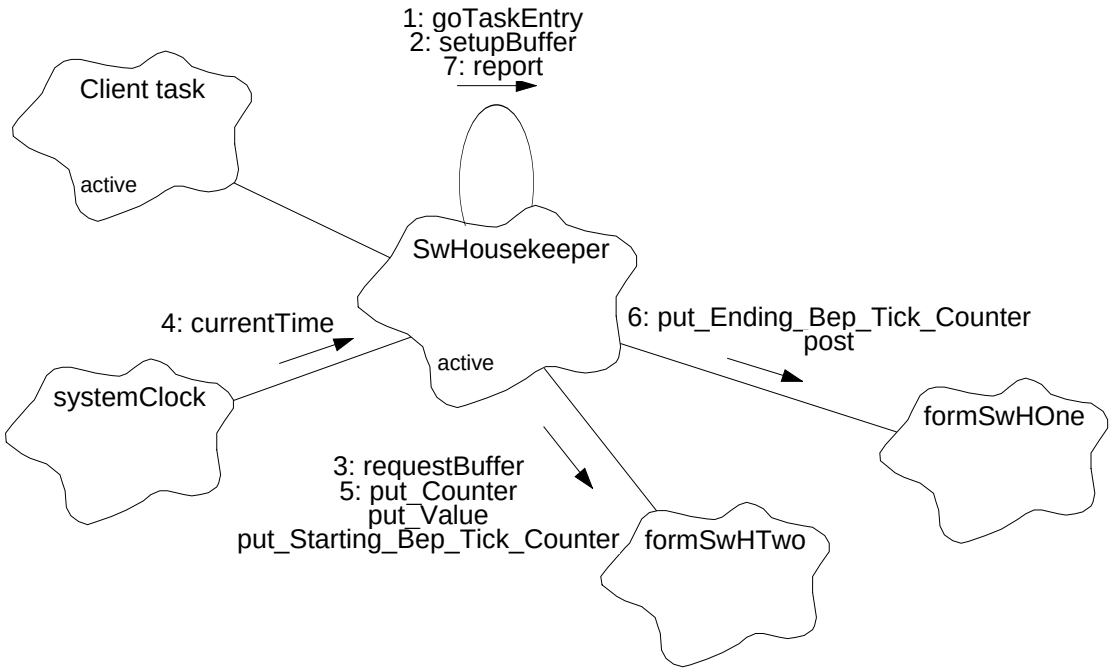
into the value using *currentPtr*->put_Counter(). It then writes the report value using *currentPtr*->put_Value(). Upon returning, the **IntrGuard** instance is destroyed, and the previous interrupt state is restored.

6. Meanwhile, the **TaskMonitor**, on its own schedule, will query the software housekeeper to determine if it is viable. During its accumulation period the housekeeper would reply using **TaskMonitor::**respond().

28.4.3 Use 2 : A system for periodically delivering the accumulated statistics

Figure 138 illustrates the operation of the Software Housekeeper packet delivery process.

FIGURE 138. Software Housekeeper Delivery of Statistics



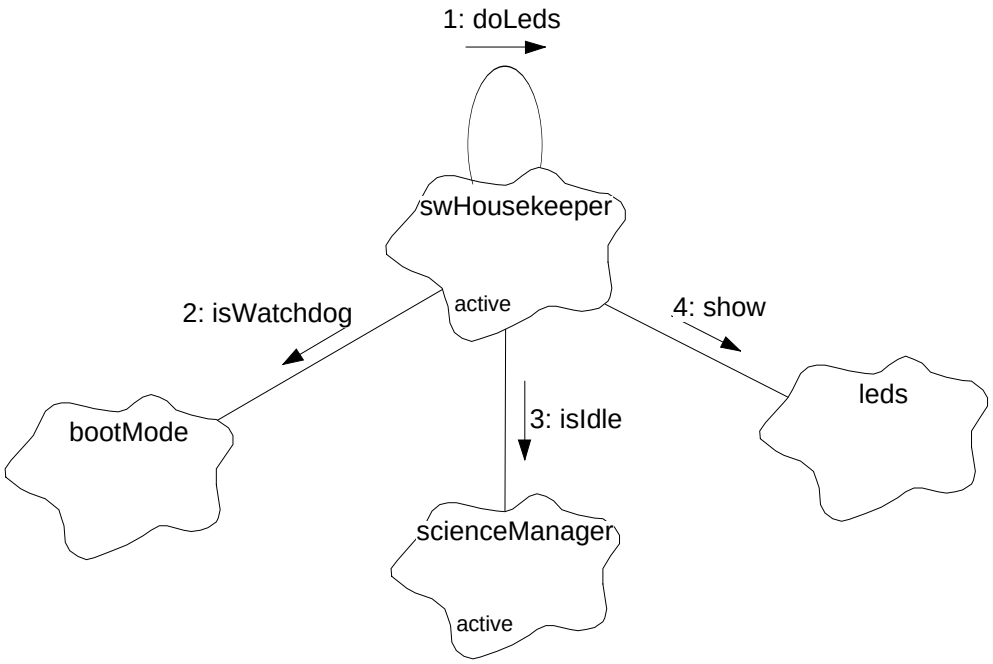
1. The software housekeeper thread, `goTaskEntry`(), associates the local pointer, *`nextPtr`* with the packet instance-in-waiting. During operation, the software housekeeper will alternate between this packet instance and the instance *`currentPtr`* associated in the constructor. The software housekeeper then enters its main task loop during which data accumulates and delivery of the housekeeping database packet is initiated.
2. At the end of the accumulation period, the housekeeper passes *`nextPtr`* to `setupBuffer`() to attempt to stage the next accumulation telemetry buffer.
3. `setupBuffer`() calls *`form`*->`requestBuffer`() to attempt to obtain a second telemetry packet buffer.
4. If the request succeeds, `setupBuffer`() calls *`systemClock`*.`currentTime`() to obtain the current BEP tick counter value (NOTE: This 1/10th second operating system timer is not the same as the science timestamp counter).
5. `setupBuffer`() then zeros the buffer's counter and value entries using *`form`*->`put_Counter`() and *`form`*->`put_Value`(). It then stores the starting integration time using *`form`*->`put_Starting_Bep_Tick_Counter`(). `setupBuffer`() supplies its caller with the counter value to delimit the end of the previous buffer's integration.

6. Once the second buffer has been obtained and initialized, `goTaskEntry`() swaps telemetry packet buffers by switching the values of *`currentPtr`* and *`nextPtr`*, through the temporary pointer, *`priorPtr`*. It then stores the ending integration BEP tick counter into the old buffer using *`priorPtr`*->`put_Ending_Bep_Tick_Counter`(). It then posts the old buffer to telemetry using *`priorPtr`*->`post`().
7. Should another buffer not be available, software statistics will continue to accumulate in the current packet buffer. The housekeeper will record the incident in the database using its own `report`() function.

28.4.4 Use 3: Indicate instrument state to software discrete telemetry (LEDs)

Figure 139 illustrates the method used by the Software Housekeeper to periodically update the software discrete telemetry levels.

FIGURE 139. Update Instrument State Indicators



1. Once per iteration of its infinite loop, the Software Housekeeper's main task function, `goTaskEntry`(), calls `doLeds`() to update the software discrete telemetry state.
2. `doLeds`() determine if the most recent reset was due to the watchdog timer using *`bootMode`*.`isWatchdog`().
3. `doLeds`() determine if a science run is underway using *`scienceManager`*.`isIdle`().
4. `doLeds`() uses the acquired boot and science state information, and the current state of the *`aPhase`* instance variable to select the appropriate LED code, and writes the code to the software discrete telemetry bits using *`leds`*.`show`().

28.5 Class SwHousekeeper

Documentation

The `SwHousekeeper` periodically initiates telemetering of accumulated statistical data provided by various software elements. During the normal course of their operation, the software tasks, which are intended to provide statistics, will call the software housekeeper and deliver the pertinent data which will be installed into the software housekeeping data structure. Periodically, the housekeeper will attempt to obtain a packet buffer for the replacement database. Failing, it will report the incident, and continue to accumulate in the current database. Succeeding, it will swap databases, initiate delivery of the acquired data to the telemetry service, and begin a new accumulation period.

Export Control: Public

Cardinality: 1

Hierarchy

Superclasses: **`Task`**

Public Interface

Operations:

```
SwHousekeeper()
goTaskEntry()
report()
```

Private Interface

Operations

```
doLeds()
intervalWait()
setupBuffer()
```

Has-A Relationships:

`Tf_Sw_Housekeeping` *`formSwHOne`*: This is an instance of a software housekeeping packet form. A packet buffer in bulk memory will be associated with this instance.

`Tf_Sw_Housekeeping` *`formSwHTwo`*: This is an instance of a software housekeeping packet form. A packet buffer in bulk memory will be associated with this instance.

`Tf_Sw_Housekeeping *`*`currentPtr`*: This pointer is used to indicate which software housekeeping packet is currently available to be filled with data

`Boolean` ***`aPhase`***: This indicates the current software discrete telemetry (LED) ping-pong state. This code toggles between ***`BoolTrue`*** and ***`BoolFalse`*** as the LEDs are periodically updated.

`const unsigned` ***`accumInterval`***: This variable contains time, in BEP timer ticks (10 per second)), over which to accumulate software housekeeping statistics. If telemetry resources are available, one housekeeping packet will be sent after each interval. The interval duration is approximately 1 minute.

`const unsigned` ***`statCount`***: This is the largest statistic code used by the housekeeper (NOTE: The class constructor initializes this value to SWH_MAX_STAT = 64. To modify this value, patch the constructor.).

Concurrency: Active

Persistence: Persistent

28.5.1 SwHousekeeper()

Public member of: **SwHousekeeper**

Arguments

unsigned *taskId*: Identifies the task being constructed.

Documentation

The SwHousekeeper() constructor initiates acquisition of the first software housekeeping statistics database packet.

Preconditions

It is expected that functions which provide a telemetry packet will be in place when this function is constructed.

Semantics

SwHousekeeper() uses setupBuffer(), which acquires the packet buffer. The Buffer MUST have been obtained, or a fatal error will be generated!

Concurrency: Guarded

28.5.2 doLeds()

Private member of: **SwHousekeeper**

Return Class: **void**

Documentation

This function blinks the software discrete telemetry bits (LEDs) to indicate the current state of the instrument. The state of the LEDs are changed on each call to this function.

Semantics

This list of housekeeping LED codes is as follows, where their values are defined in the ACIS Software IP&CL, MIT 36-53204.0204:

LED_WD_SCIENCE_A
LED_WD_SCIENCE_B
LED_WD_IDLE_A
LED_WD_IDLE_B
LED_RUN_SCIENCE_A
LED_RUN_SCIENCE_B
LED_RUN_IDLE_A
LED_RUN_IDLE_B

Use *bootMode*.isWatchdog() to determine if watchdog cause the most recent reset. If so, select the LED_WD_* set of LED codes, otherwise, select the LED_RUN_*.

Use *scienceManager*.isIdle() to determine if science is performing a run. If so, select the LED_*_SCIENCE_* set of codes, otherwise, select the LED_*_IDLE_* set.

Toggle between LED_*_*_A and LED_*_*_B on each call to the function. Use the *aPhase* instance variable to determine which blink state to use. If *aPhase* is *BoolFalse*, select the LED_*_*_B set of LEDs, and set *aPhase* to *BoolTrue*. If *aPhase* is already *BoolTrue*, use the LED_*_*_A set of codes, and set *aPhase* to *BoolFalse*.

Use the selection code to lookup the corresponding LED code, and write the code using **Leds**::show().

.

Concurrency: Guarded

28.5.3 goTaskEntry()

Public member of: **SwHousekeeper**

Return Class: **void**

Documentation

goTaskEntry() is the initiation point for the software housekeeper. On a regular basis, it initiates dispatch of the accumulated housekeeping statistics being telemetered.

Preconditions

The constructor for SwHousekeeper() has initiated the first packet.

Semantics

On start-up, goTaskEntry() associates a pointer with the second **Tf_Sw_Housekeeping** software housekeeping packet. It then enters a FOREVER loop, and remains inactive in intervalWait() for a fixed time period (*accumInterval*) during which statistics are accumulated into the telemetry buffer. When activated, it initiates an attempt to obtain a replacement empty telemetry packet buffer using setupBuffer(). If it is successful, it will switch packet buffer pointers and begin accumulating into the fresh packet buffer database, store the ending integration time into the prior buffer, and post it to telemetry using **TlmForm::**post(). Otherwise, it will record the condition using **SwHousekeeper::**report() to record a skipped software housekeeping packet delivery (statistic), before continuing accumulation in the same packet during the ensuing interval. At the end of each iteration, goTaskEntry() calls doLeds() to update the software discrete telemetry codes.

Post Conditions

This procedure never returns.

Concurrency: Guarded

28.5.4 intervalWait()

Private member of: **SwHousekeeper**

Return Class: **void**

Documentation

`intervalWait`() idles the process for a predefined period. During this interval, housekeeping reports are filed by various entities. Also, during this accumulation period, `intervalWait`() will respond to `taskMonitor`() interrogations.

Semantics

`intervalWait`() idles in `waitForEvent`() while it waits for the monitor (are you alive) interrogation events or for *`accumInterval`* number of Science Frame pulse timing tick events. It will respond to the interrogation and will count the requisite number of pulses before returning.

Concurrency: Guarded

28.5.5 report()

Public member of: **SwHousekeeper**

Return Class: **void**

Arguments

SwStatistic *statisticId*:: Identifies the statistic being recorded.

unsigned *value*:: Contains a (possibly irrelevant) associated value.

Documentation

report() is the vehicle provided to accumulate the referenced statistics which the software housekeeper will deliver.

Semantics

First, the function tests *currentPtr* and returns if it is NULL. If *currentPtr* is not NULL, it verifies that the *statisticId* is within range (i.e. be less than *statCount*). If not, it sets the statistic id to SWSTAT_SWHOUSE_RANGE. It then disables by declaring an **IntrGuard** instance. It then uses *currentPtr*->get_Counter() and *currentPtr*->put_Counter() to read, increment and write the counter indexed by *statisticId*. It then uses *currentPtr*->put_Value() to store *value*. Upon returning, the **IntrGuard** destructor restores the original interrupt enable state.

Concurrency: Synchronous

28.5.6 setupBuffer()

Private member of: **SwHousekeeper**

Return Class: **Boolean**

Arguments

Tf_Sw_Housekeeping* *form*:: Points to telemetry formatter

unsigned& *starttick*:: Used to return integration start time

Documentation

This function sets up a telemetry packet buffer for the telemetry form, pointed to by *form*. If a buffer is successfully obtained, the function sets *starttick* to the starting integration BEP timer-tick value and returns *BoolTrue*. If a telemetry packet buffer is not available at the time of the call, the function returns *BoolFalse*.

Semantics

Call *form*->requestBuffer(). If successful, call *systemClock*.currentTime() to obtain the current timestamp, zero the statistics counters and values using *form*->put_Counter() and *form*->putValue(). Store the starting timer tick using *form*->put_Starting_Bep_Tick_Counter() and return *BoolTrue*. If *form*->requestBuffer() fails to obtain a packet buffer, return *BoolFalse*.

Concurrency: Guarded

29.0 FatalError (36-53243 01)

29.1 Purpose

The Fatal Error class provides notification that an irrecoverable condition exists and controls an expeditious watchdog CPU reset.

29.2 Uses

Any of the processes or functions may use Fatal Error. Normally, requests for this service are a result of some function encountering a illegal value or condition.

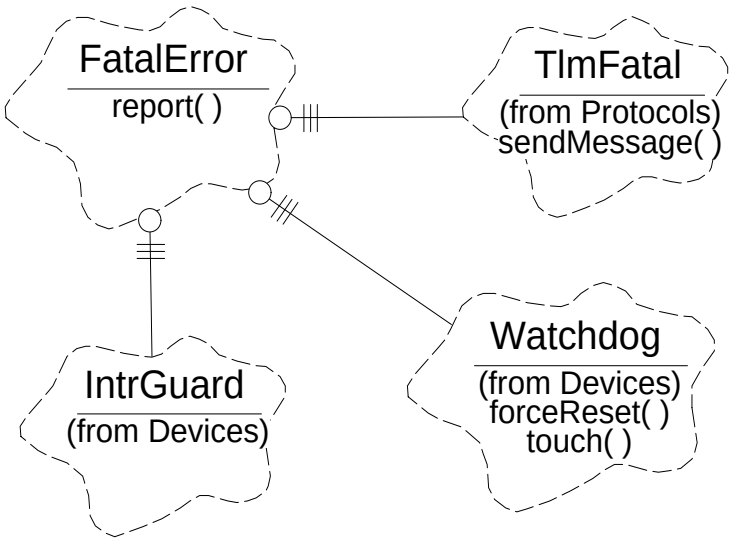
Specifically it provides the following features:

Use 1:: Initiates a panic message which identifies the fault encountered
Use 2:: Forces a system reset.

29.3 Organization

Figure 140 illustrates the relationship between the classes used by Fatal Error.

FIGURE 140. Fatal Error Class Relationships



Fatal Error uses ***Devices***, and, ***Protocols,*** class categories.

IntrGuard - This class is provided by the ***Devices*** class category, and is used to prevent interrupts from interfering with **FatalErrors** activities.

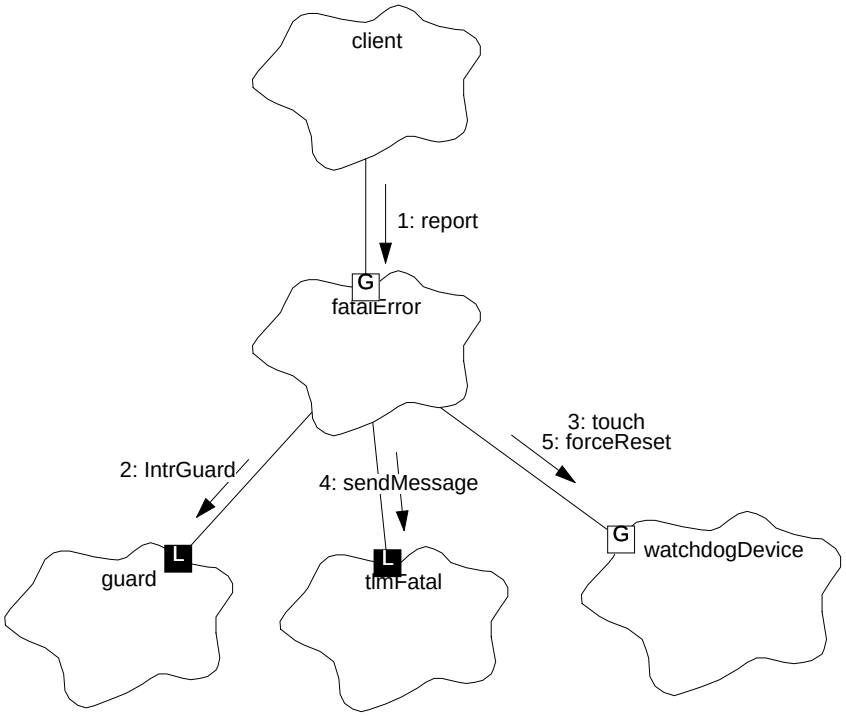
TlmFatal - This class is provided by the ***Protocols*** class category and is responsible for insertion of the data into the packet and for initiating delivery of the panic message.

Watchdog - This class is provided by the ***Devices*** class category and is responsible for resetting the hardware watchdog timer.

29.4 Scenarios

The **FatalError**.report() 1: may be called by any active process. It is delivered a value identifying the error encountered, and a second argument which provides further information. A call to **FatalError**.report() never returns.

FIGURE 141. Fatal Error Scenario



29.4.1 Use 1: Deliver Panic Message

FatalError.report() invokes **IntrGuard**.guard 2: which disables interrupts. **FatalError** will then touch() 3: the watchdog providing sufficient time to telemeter the error message. Failure to complete the following steps will result in the watchdog resetting when its regular interval completes since the disabled interrupts will keep the taskMonitor() from touching the watchdog.

`report`() then delivers the information to the **`TlmFatal`** form using its `sendMessage`() 4: function which installs the arguments provided by the client into a packet buffer, and hands it off to **`TlmManager`**`.sendPanic`() for delivery (not shown). `sendPanic`() attempts to allow an outgoing message to complete before resetting the telemetry device, handing off the message, and idling for a nominal interval before returning.

29.4.2 Use 2: Handle Watchdog

The **`Watchdog`**`.forceReset`() 5: is used to reset that device to the shortest interval, and then busy loops until the CPU is reset.

29.5 Class Fatal Error

Documentation

`FatalError` provides the ability to issue a fatal error telemetry report, then hastens the hardware watchdog's reset of the system.

Export Control: Public

Cardinality: 1

Hierarchy

Superclasses: **none**

Public Interface

Operations:

`FatalError`()

`report()`

Concurrency: Synchronous

Persistence: Transient

29.5.1 FatalError()

Public member of: **FatalError**

Return Class: **void**

Arguments: **none**

Documentation

This constructor initializes the `FatalError` instance.

Concurrency: Sequential

29.5.2 report()

Public member of: **FatalError**

Return Class: **void**

Arguments:

enum Fatal_Code `errorNum`
unsigned `opInfo`

Documentation

`report`() provides the means to control interrupts, deliver a panic message and set the shortest ***`Watchdog`*** interval to immediately reset the CPU.

Semantics

When a client activates `report`(), it disables interrupts, touches the watchdog timer, initiates installation of the arguments provided into the packet using **`TlmFatal`**.`sendMessage`() which hands it off to the ***`Telemetry Manager`*** for delivery. `report`() invokes **`Watchdog`**.`forceReset`() which will cause an immediate watchdog reset.

Postconditions

This function Never returns.

Concurrency: Sequential

30.0 System Configuration Classes (36-53238 A)

30.1 Purpose

The purpose of the System Configuration classes is to manage the suite of software-controllable hardware settings within the instrument.

30.2 Uses

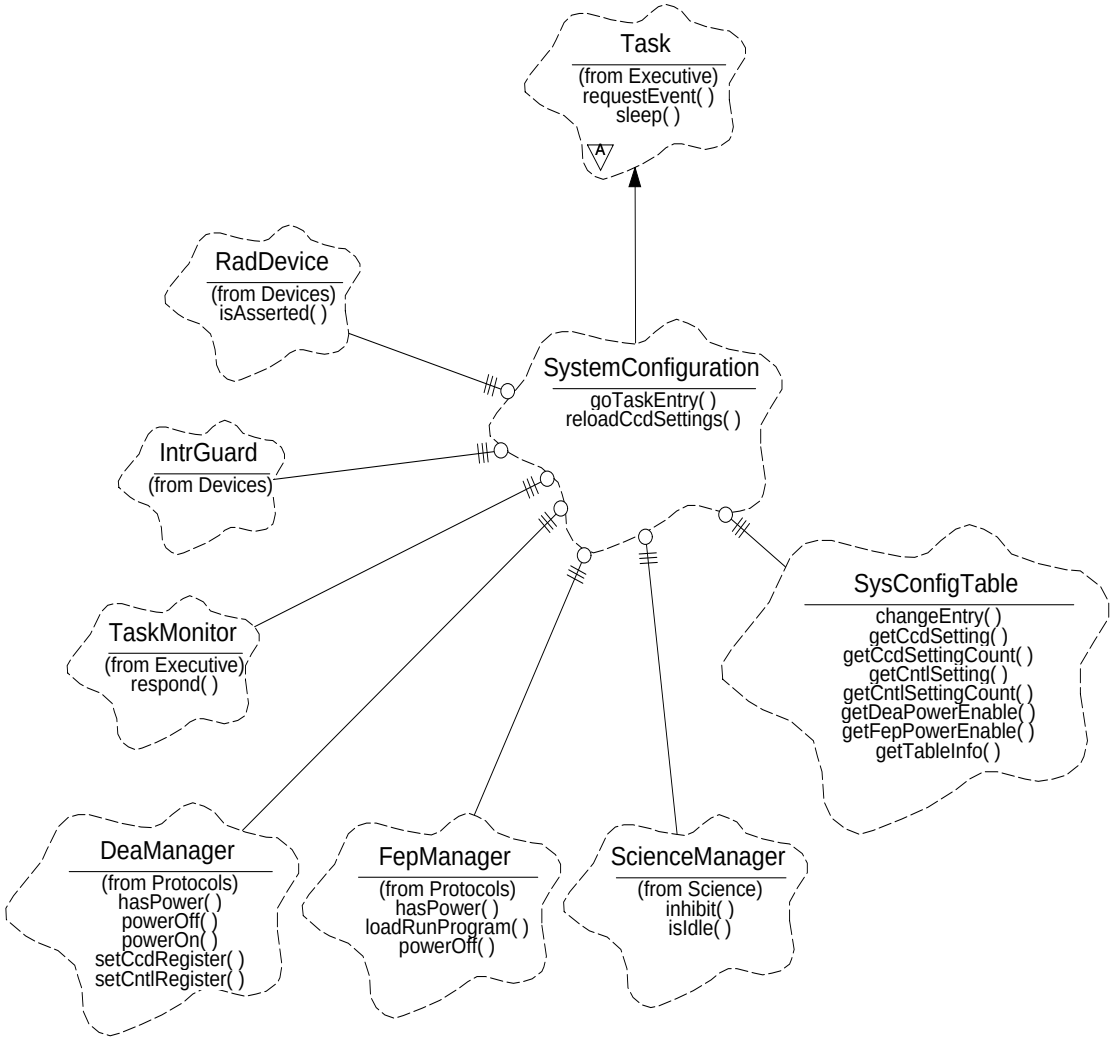
The following lists the use of the System Configuration Classes:

Use 1:: Update settings within the system's table
Use 2:: Update DEA and FEP power selections
Use 3:: Load updated settings into the DEA interface controller
Use 4:: Power off the DEA if the radiation monitor is asserted
Use 5:: Re-enable DEA power once the radiation monitor de-asserts
Use 6:: Re-load settings into a DEA CCD controller

30.3 Organization

Figure 142 illustrates the class relationships used by the System Configuration Classes. These class include the **`SysConfigTable`** class, which maintains the current list of settings, and the **`SystemConfiguration`** class, which is a task which reloads settings into the hardware and monitors the radiation flag.

FIGURE 142. System Configuration Class Relationships



`SysConfigTable`- This class represents the collection of system configuration parameters within the instrument. It provides a function which modifies an entry in the table (`changeEntry`), and provides a function which returns the address and length of the table for purposes of dumping the table to telemetry (`getTableInfo`). This class provides services used by the **`SystemConfiguration`** class to get the desired power settings for each of the DEA and FEP boards (`getDeaPowerEnable`, `getFepPowerEnable`), to fetch a desired DEA CCD Controller or Interface Controller register setting from the table (`getCcdSetting`, `getCntlSetting`, respectively), and to determine the total number of settings for the CCD Controller or Interface Controller (`getCcdSettingCount`., `getCntlSettingCount`, respectively).

`SystemConfiguration`- This class is a subclass of ***`Executive`***::**`Task`**, and is responsible for maintaining the hardware configuration settings and reacting to the radia-

tion monitor. This class provides a main task entry function (`goTaskEntry`) and a public function used to reload all settings into a CCD Controller (`reloadCcdSettings`).

`Task`- This class is supplied by the ***`Executive`*** class category. It represents and controls an active running task. The **`SystemConfiguration`** class inherits from this class, and uses the class's functions to relinquish control for a period of time (`sleep`), and to detect queries from the TaskMonitor (`requestEvent`).

`RadDevice`- This class is supplied by the ***`Devices`*** class category. It is responsible for providing access to the radiation monitor flag (`isAsserted`).

`IntrGuard`- This class is supplied by the ***`Devices`*** class category and is responsible for disabling and re-enabling interrupts.

`TaskMonitor`- This class is supplied by the ***`Executive`*** class category, and is responsible for periodically polling each task in the instrument. When polled, the **`SystemConfiguration`** task responds using this classes member function (`respond`).

`DeaManager`- This class is supplied by the ***`Protocols`*** class category, and is responsible for arbitrating access to the DEA hardware interface. The **`SystemConfiguration`** class uses this class to disable power to the DEA boards (`powerOff`), to query the current power state of the boards (`hasPower`), to enable power to the boards (`powerOn`), and to write to the CCD Controller and Interface Controller boards registers (`setCcdRegister`, `setCntlRegister`, respectively).

`FepManager`- This class is supplied by the ***`Protocols`*** class category, and is responsible for managing access to the Front End Processors. The **`SystemConfiguration`** class uses this class to query the current power state of each of the FEPs (`hasPower`), to disable power to a FEP (`powerOff`), and to power on a FEP and load a default program (`loadRunProgram`).

`ScienceManager`- This class is supplied by the ***`Science`*** class category, and is responsible for managing overall science processing. The **`SystemConfiguration`** class uses this class to detect when a science run is complete (`isIdle`), and to abort a run in-progress and inhibit further runs (`inhibit`).

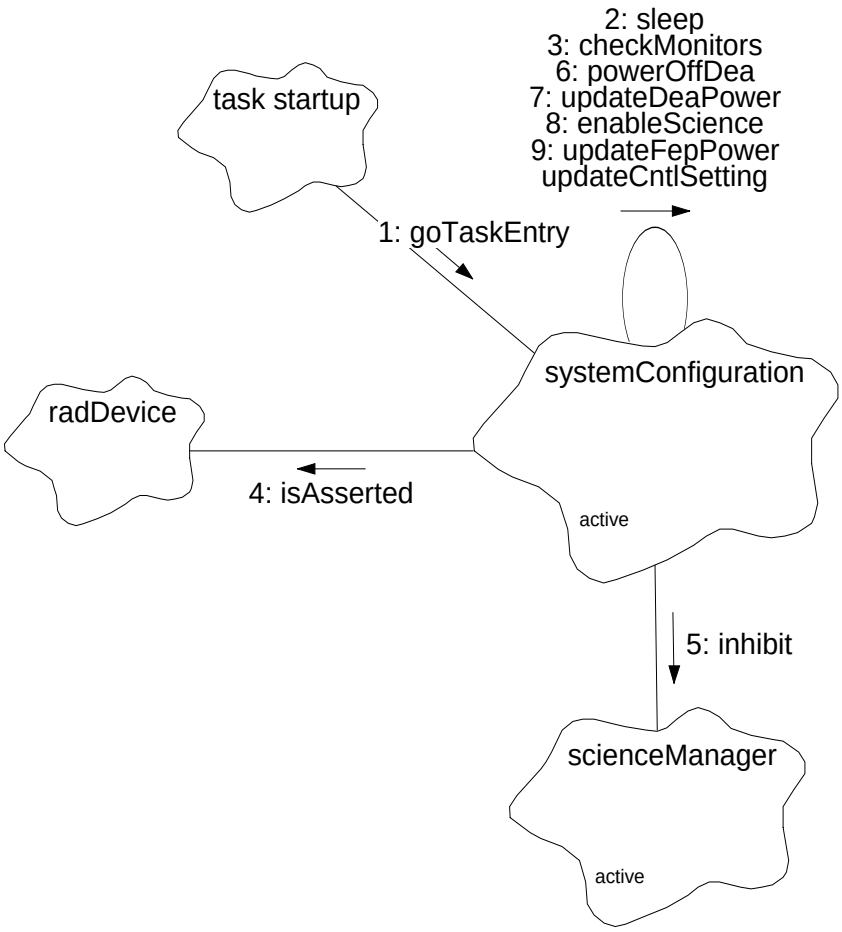
30.4 Scenarios

30.4.1 Overall SystemConfiguration task operation

The **`SystemConfiguration`** task consists of a main polling loop, contained within `goTaskEntry`(), which uses the **`RadDevice`** class to determine when the radiation monitor is asserted.

Figure 143 illustrates the main operations performed on each iteration of this loop, and the interactions with the Radiation Monitor interrupt.

FIGURE 143. SystemConfiguration task polling loop



1. During system startup, the executive starts the *`systemConfiguration`* task using *`systemConfiguration`*.`goTaskEntry`(). The task function `goTaskEntry`() then enters its infinite polling loop.
2. At the top of its loop, `goTaskEntry`() calls `sleep`() to allow other tasks to run.
3. It then checks for, and responds to queries from the *`taskMonitor`*, and checks the current state of the radiation monitor using `checkMonitors`().

4. It then checks if the radiation monitor has been asserted, using radDevice.isAsserted(). Assume for this example, that the monitor has been asserted.
5. Once the task detects the assertion of the monitor, it tells science to end its current run and delay the start of any subsequent runs until the radiation condition subsides, by passing *BoolTrue* to *scienceManager*.inhibit().
6. The task then shuts off power to all of the DEA's CCD-controller boards, using powerOffDea().
7. If the radiation monitor is not asserted, the task then updates any changes to the DEA power settings using updateDeaPower().
8. If the monitor had previously been asserted, the task then enables science runs using enableScience(), which then passes *BoolFalse* to *scienceManager*.inhibit() to perform the enable (and possibly re-start a previously shutdown science run).
9. On each iteration, the task updates the FEP power settings using updateFepPower(), and updates the DEA Interface Controller settings using updateCntlSettings().

30.4.2 Use 1: Update settings within the system's table

To modify an entry in the system configuration table, the client passes the entry index and value to *sysConfigTable*.changeEntry(), which then modifies the indexed value and sets the corresponding changed flag to *BoolTrue*. Later, the *systemConfiguration* task will pick up the modification, adjust the corresponding DEA hardware setting or power selection as described in the subsequent scenarios and sets the changed flag to *BoolFalse*.

The final layout of the table is TBD. Table 26 illustrates the current layout:

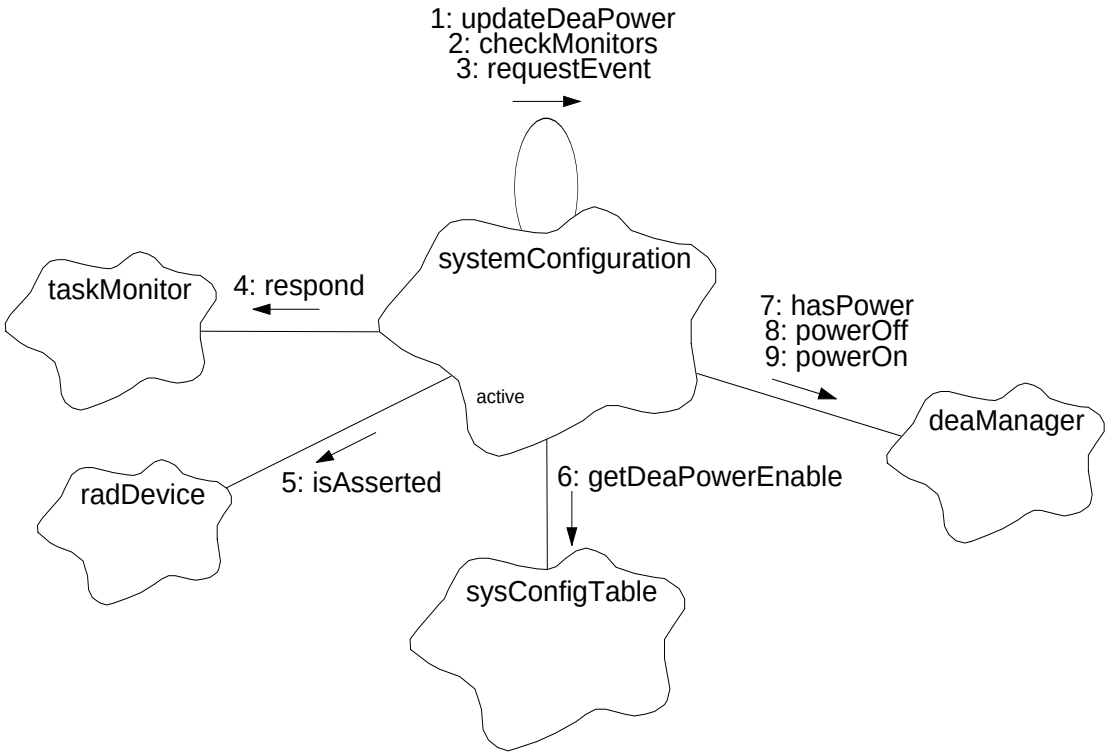
TABLE 26. System Configuration Table Layout

Index	Name	Quantity	Description
0	DEA Power	1	This is a bit-field, indicating which CCD-controllers should be powered. Each bit is indexed by CCD Id (i.e. bit 0 corresponds to CCD_I0). If a bit is '1', then the CCD controller should be on, otherwise, it should be off. Unused bits in the field should be set to zero.
1	FEP Power	1	This is a bit-field, indicating which FEPs should be powered. Each bit is indexed by FEP Id (i.e. bit 0 corresponds to FEP_0). If a bit is '1', then the FEP should be on, otherwise, it should be off. Unused bits in the field should be set to zero.
2..65	DEA Interface Controller Settings	64	This is an array of interface board settings. The assignments are TBD.
66+ (64 * *ccdId*) + *settingId*	DEA CCD Controller Settings	10 * 64	This is a set of arrays of CCD Controller settings. The index formula indicates the offset to a particular setting, indicated by *settingId* for a particular CCD Controller, indicated by *ccdId*. The register assignments are TBD.

30.4.3 Use 2: Update DEA and FEP power selections

Figure 144 illustrates the case where the **`SystemConfiguration`** updates the DEA power settings.

FIGURE 144. Update DEA Power Settings



1. The main polling loop of the *`systemConfiguration`* task determines that the radiation monitor is off. It takes the opportunity to update any DEA power settings which have changed by calling `updateDeaPower`().
2. `updateDeaPower`() consists of two loops. The first loop powers off any DEA boards which have been disabled. The second loop then powers on any boards which have been enabled (NOTE: The settings for a given CCD controller are re-loaded at the onset of the next science run). At the top of each loop iteration, `updateDeaPower`() calls `checkMonitors`() to ensure that the *`taskMonitor`* and radiation monitors are responded to in a timely fashion.
3. `checkMonitors`() first checks to see if a query from the *`taskMonitor`* is outstanding, using the inherited function, **`Task`**::`requestEvent`().
4. If a query is pending, `checkMonitors`() replies to the query using *`taskMonitor`*.`respond`().
5. `checkMonitors`() then checks to see if the radiation monitor is active by calling `isRadiationOn`() (not shown). `isRadiationOn`() then calls *`radDevice`*.`isAsserted`() to test the hardware radiation monitor flag. If the flag is

asserted, then checkMonitors() returns, indicating that the current update operation should be aborted. If the radiation monitor is inactive, then the current operation may proceed. Assume for the remainder of the scenario that the monitor is inactive.

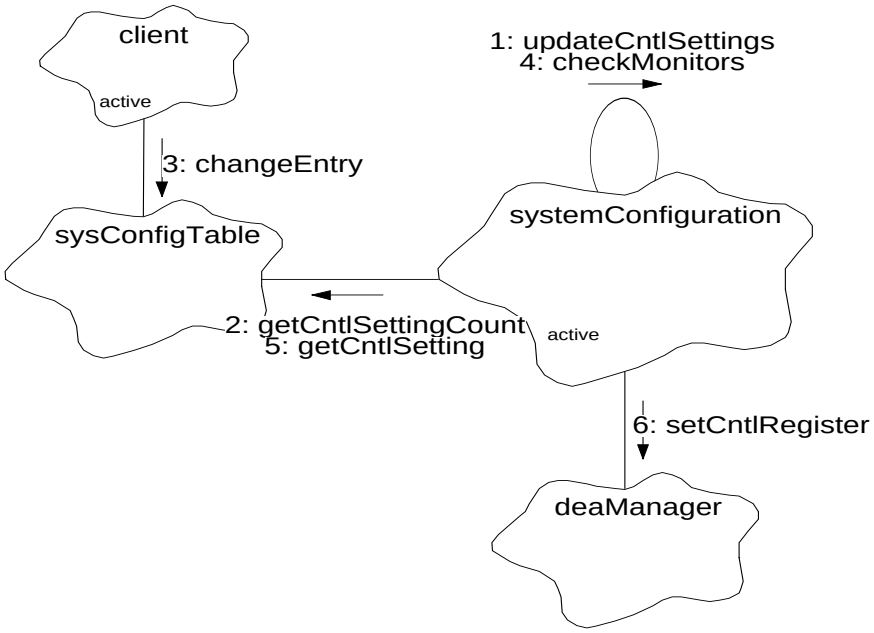
6. updateDeaPower() then obtains the desired power setting for a DEA board using *sysConfigTable*.getDeaPowerEnable().
7. updateDeaPower() then compares this with the current power state of the board, as indicated by *deaManager*.hasPower().
8. If the configuration indicates that the board should be off, but it is on, updateDeaPower() shuts off power to the board using *deaManager*.powerOff().
9. Once all of the boards which should be off have been disabled, updateDeaPower() checks for boards which should be turned on. If a board's power is off when it should be on, updateDeaPower() enables power to the board using *deaManager*.powerOn().

The scenario for powering FEPs is similar to the above scenario except that the desired power settings are obtained using *sysConfigTable*.getFepPowerEnable(), and the queries and power control functions (hasPower, powerOff, powerOn) are issued to the *fepManager*, rather than the *deaManager*. Since changes in FEP power are driven by power-consumption and thermal concerns, power settings to the FEPs are always updated, regardless of the state of the radiation monitor.

30.4.4 Use 3: Load updated settings into the DEA interface controller

Figure 145 illustrates the steps used by the **`SystemConfiguration`** class to update modified register settings in the DEA interface controller board.

FIGURE 145. Load updated DEA settings

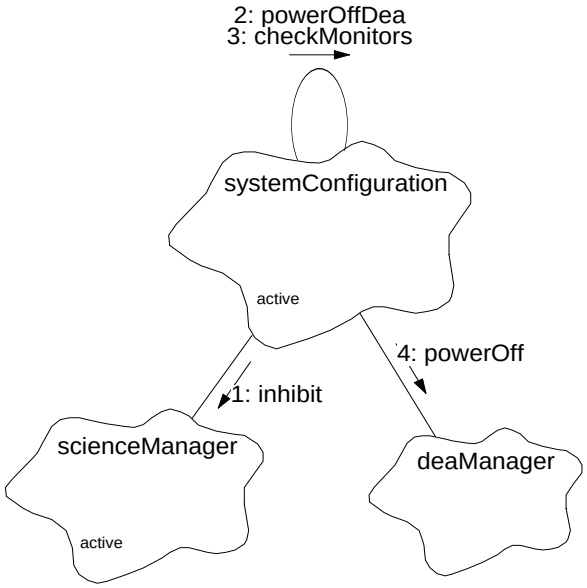


1. The *`systemConfiguration`*'s polling loop calls `updateCntlSettings`() to update any DEA interface controller register values which have changed.
2. `updateCntlSettings`() obtains the total number of settings for the interface board using *`sysConfigTable`*.`getCntlSettingCount`(). It then enters its main loop to update the collection of settings.
3. Meanwhile, a client object may make changes to the collection of settings using *`sysConfigTable`*.`changeEntry`(). If a change is made to a setting prior to it's being processed by `updateCntlSettings`(), it will be loaded into the hardware on this invocation of `updateCntlSettings`(). If, however, a change is made after it has been passed over by `updateCntlSettings`(), it won't be loaded into the DEA until the next iteration of the *`systemConfiguration`* task's polling loop.
4. At the start of each iteration of the loop, `updateCntlSettings`() calls `checkMonitors`() to respond to queries from the *`taskMonitor`*.
5. `updateCntlSettings`() obtains the desired value for a given DEA board's register, whether or not it is used by the hardware, and whether or not the desired setting has changed since the last update, using *`sysConfigTable`*.`getCntlSetting`().
6. If the register setting is used, and has changed, `updateCntlSettings`() loads the value into the DEA board using *`deaManager`*.`setCntlRegister`().

30.4.5 Use 4: Power off the DEA if the radiation monitor is asserted

Figure 146 illustrates the steps used by the **`SystemConfiguration`** class to shutdown the DEA when the radiation monitor has been asserted.

FIGURE 146. DEA shutdown due to radiation monitor assertion



1. When the *`systemConfiguration`* task's polling loop detects that the radiation monitor has gone off, it instructs the *`scienceManager`* to stop its current run, and delay the onset of any subsequent runs, by passing *`BoolTrue`* to *`scienceManager`*.`inhibit`().
2. It then immediately shuts off power to all of the DEA's CCD-controller boards using `powerOffDea`().
3. `powerOffDea`() iterates through each board. At top of each iteration, `powerOffDea`() calls `checkMonitors`() to respond to any queries from the *`taskMonitor`*. Since it already knows that the radiation monitor has been asserted, `powerOffDea`() ignores the return value from `checkMonitors`().
4. As part of its processing loop, `powerOffDea`() then shuts off power to each DEA CCD-controller using *`deaManager`*.`powerOff`().

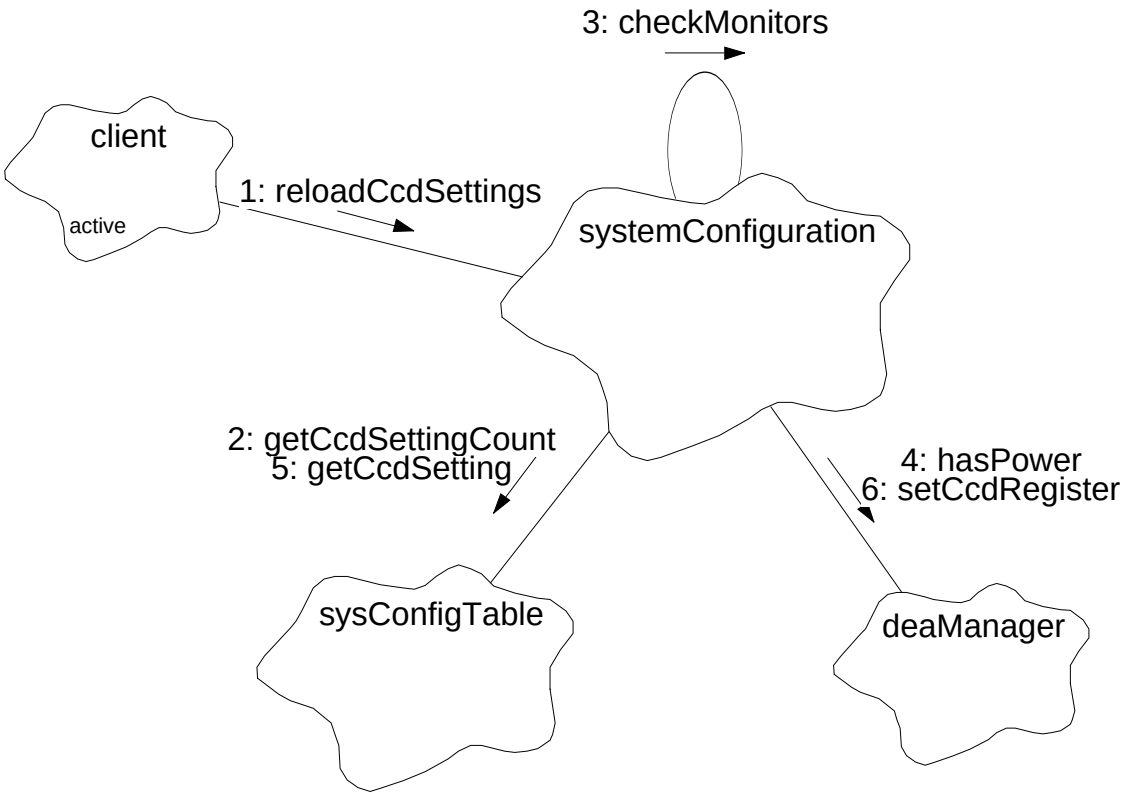
30.4.6 Use 5: Re-enable DEA power once the radiation monitor de-asserts

Once the radiation monitor subsides, the *systemConfiguration*'s polling loop calls updateDeaPower() to power on all of the required DEA boards (see Section 30.4.3). Their respective settings are re-loaded at the onset of the next science run. It then enables science by passing *BoolFalse* to *scienceManager*.inhibit().

30.4.7 Use 6: Re-load settings into a DEA CCD controller

The settings for a given CCD Controller are re-loaded when the CCD is about to be used for a science run. Its settings are loaded as part of the setup for the run. Figure 147 illustrates the steps used by the SystemConfiguration to load the settings for a particular CCD.

FIGURE 147. Re-load CCD Controller Settings



1. The client (*scienceManager*) attempts to re-load the settings for a particular CCD controller using *systemConfiguration*.reloadCcdSettings().
2. reloadCcdSettings() obtains the number of settings used for a CCD controller using *sysConfigTable*.getCcdSettingCount().
3. reloadCcdSettings() then enters loading loop. On each iteration, reloadCcdSettings() calls checkMonitor() to test for and respond to queries from the **TaskMonitor**.

4. `reloadSettings`() checks that the selected CCD controller was commanded to be powered on using *`deaManager`*.`hasPower`(). If the board was not powered on, then the function immediately returns.
5. If the board has power, `reloadCcdSettings`() obtains the next setting using *`sysConfigTable`*.`getCcdSetting`().
6. It loads the setting into the CCD Controller using *`deaManager`*.`setCcdRegister`().

30.5 Class SysConfigTable

Documentation:

This class acts as a repository for the system configuration settings.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
SysConfigTable()
changeEntry()
getCcdSetting()
getCcdSettingCount()
getCntlSetting()
getCntlSettingCount()
getDeaPowerEnable()
getFepPowerEnable()
getTableInfo()
```

Protected Interface:

Operations: `getSetting()`

Private Interface:

Has-A Relationships:

`Boolean` *`inuseCcdMap`*[64 (TBD)]: This is an array of in-use flags indicating whether a particular DEA CCD-controller board register is used by the hardware or not. The array is indexed by register index. If an entry is *`BoolFalse`*, then the corresponding setting is not used by the system. If an entry is *`BoolTrue`*, then the setting entry is valid.

`Boolean` *`inuseCntlMap`*[64 (TBD)]: This is an array of in-use flags indicating whether a particular DEA interface board register is used by the hardware or not. The array is indexed by register index. If an entry is *`BoolFalse`*, then the corresponding setting is not used by the system. If an entry is *`BoolTrue`*, then the setting entry is valid.

Boolean *settingChanged*[2818 (TBD)]: This is an array of flags which indicate which values have been changed since the last read. *BoolFalse* indicates that the value hasn't changed since the last query, and *BoolTrue* indicates that the value has changed. (TBD: use bit-field)

unsigned short *setting*[2818 (TBD)]: This is the array of configuration setting values.

Concurrency: Guarded

Persistence: Persistent

30.5.1 SysConfigTable()

Public member of: **`SysConfigTable`**

Documentation:

This function is the constructor for the **`SysConfigTable`** class.This function initializes the in-use tables, and flags all settings as changed.

Concurrency: Sequential

30.5.2 changeEntry()

Public member of: **`SysConfigTable`**

Return Class: **`void`**

Arguments:

`unsigned` *`item`*
`unsigned` *`value`*

Documentation:

This function modifies the configuration entry, setting the entry indicated by *`item`* to the passed *`value`*.

Concurrency: Synchronous

30.5.3 getCcdSetting()

Public member of: **SysConfigTable**

Return Class: **Boolean**

Arguments:

CcdId *ccdid*
unsigned *regid*
unsigned short& *value*
Boolean& *changed*

Documentation:

This function reads the desired setting for the DEA CCD Controller register. The CCD controller is indicated by *ccdid*, and the register setting is indexed by *regid*. The function returns the desired setting in *value* and resets the modification flag corresponding to the entry. If the value has changed since the last call, the function sets *changed* to *BoolTrue*. If the value hasn't changed since the last time it was fetched, *changed* contains *BoolFalse*. If the register is not used by the DEA hardware, the function returns *BoolFalse*. If it is a valid hardware register, the function returns *BoolTrue*. (NOTE: This scheme is to allow for late changes to the DEA register settings without impacting the software design).

Concurrency: Synchronous

30.5.4 getCcdSettingCount()

Public member of: **SysConfigTable**

Return Class: **unsigned**

Documentation:

This function returns the fixed number of CCD controller settings in the table.

Concurrency: Synchronous

30.5.5 getCntlSetting()

Public member of: **SysConfigTable**

Return Class: **Boolean**

Arguments:

unsigned *regid*
unsigned short& *value*
Boolean& *changed*

Documentation:

This function reads the desired setting for the DEA Interface Controller register. The register setting is indexed by *regid*. The function returns the desired setting in the value and resets the changed flag corresponding to the entry. If the value has changed since the last call, the function sets *changed* to *BoolTrue*. If the value hasn't changed since the last time it was fetched, *changed* contains *BoolFalse*. If the register is not used by the DEA hardware, the function returns *BoolFalse*. If it is a valid hardware register, the function returns *BoolTrue*. (NOTE: This scheme is to allow for late changes to the DEA register settings without impacting the software design).

Concurrency: Synchronous

30.5.6 getCntlSettingCount()

Public member of: **SysConfigTable**

Return Class: **unsigned**

Documentation:

This function returns the fixed number of Interface controller settings in the table.

Concurrency: Synchronous

30.5.7 getDeaPowerEnable()

Public member of: **SysConfigTable**

Return Class: **Boolean**

Arguments:

DeaBoardId *boardid*

Documentation:

This function returns whether or not the DEA board, indicated by *boardid*, is configured to be powered on. If the board should not be powered on, this function returns *BoolFalse*. If the board should be powered on when the radiation monitor is not active, the function returns *BoolTrue*.

Concurrency: Synchronous

30.5.8 getFepPowerEnable()

Public member of: **SysConfigTable**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function indicates whether or not the FEP, indicated by *fepid*, should be powered on. If the function returns *BoolFalse*, the FEP should be off. If the function returns *BoolTrue*, the FEP should be on.

Concurrency: Synchronous

30.5.9 getSetting()

Protected member of: **SysConfigTable**

Return Class: **void**

Arguments:

unsigned *item*
unsigned short& *value*
Boolean& *changed*

Documentation:

This function reads and resets the changed flag for the indicated *item*. The fetched item is returned in *value*. If the value has changed since the last time it was fetched, the function sets *changed* to *BoolTrue*, otherwise, *changed* contains *BoolFalse*.

Concurrency: Synchronous

30.5.10 getTableInfo()

Public member of: **SysConfigTable**

Return Class: **void**

Arguments:

const unsigned*& *addr*
unsigned& *wordcnt*

Documentation:

This function returns the address and word length of the system configuration table. Upon return, *addr* points to the start of the system configuration table, and *wordcnt* contains the number of 32-bit words in the table.

Concurrency: Synchronous

30.6 Class SystemConfiguration

Documentation:

This class manages the system configuration of the instrument. It is responsible for monitoring the radiation flag, maintaining the power settings to the DEA and FEP, and updating DEA register settings.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **Task**

Implementation Uses:

SysConfigTable
RadDevice
TaskMonitor
FepManager
DeaManager
ScienceManager
IntrGuard

Public Interface:

Operations:
```
SystemConfiguration()
goTaskEntry()
reloadCcdSettings()
```

Protected Interface:

Operations:
```
checkMonitors()
enableScience()
isRadiationOn()
powerOffDea()
updateCcdSettings()
updateCntlSettings()
updateDeaPower()
updateFepPower()
```

Private Interface:

Has-A Relationships:.

`const unsigned` *`pollSleep`*: This contains the number of ticks the task should sleep on each iteration.

`Boolean` *`radiationAssertFlag`*: This flag indicates that the radiation monitor has been asserted.

Concurrency: Active

Persistence: Persistent

30.6.1 SystemConfiguration()

Public member of: **`SystemConfiguration`**

Arguments:

`unsigned` *`taskid`*

Documentation:

This is the constructor of the system configuration class. *`taskid`* is the RTX task id to use for the task. This function first initializes the parent **`Task`**. It then initializes its instance variables.

Concurrency: Sequential

30.6.2 checkMonitors()

Protected member of: **`SystemConfiguration`**

Return Class: **`Boolean`**

Documentation:

This function polls the radiation monitor and the task monitor. If the task monitor is querying the task, the function responds to the query. If the radiation monitor is off, the function returns *`BoolFalse`*. If the monitor is on, the function returns *`BoolTrue`*.

Concurrency: Synchronous

30.6.3 enableScience()

Protected member of: **`SystemConfiguration`**

Return Class: **`void`**

Documentation:

This function enables science activities if the radiation monitor is not active.

Concurrency: Synchronous

30.6.4 goTaskEntry()

Public member of: **SystemConfiguration**

Return Class: **void**

Documentation:

This is the main loop of the system configuration task. This loop sleeps for *pollSleep* timer ticks. It then processes *taskMonitor* queries, changes to the radiation monitor flag, and updates any changed power or board settings, if allowed by the radiation monitor and science.

Concurrency: Synchronous

30.6.5 isRadiationOn()

Protected member of: **SystemConfiguration**

Return Class: **Boolean**

Documentation:

This function polls the radiation monitor. If the monitor is asserted the function returns *BoolTrue*. If the monitor is off, the function returns *BoolFalse*.

Concurrency: Synchronous

30.6.6 powerOffDea()

Protected member of: **SystemConfiguration**

Return Class: **void**

Documentation:

This function shuts down power to all of the DEA's CCD-controller boards.

Concurrency: Synchronous

30.6.7 reloadCcdSettings()

Protected member of: **SystemConfiguration**

Return Class: **Boolean**

Arguments:

CcdId *ccdid*

Documentation:

This function reloads all settings to the DEA CCD controller board indicated by *ccdid*. Currently, the function always returns *BoolTrue*.

Concurrency: Synchronous

30.6.8 updateCcdSettings()

Protected member of: **SystemConfiguration**

Return Class: **Boolean**

Documentation:

This function updates any settings which have changed in any of the powered DEA CCD controllers. It returns *BoolTrue* if it completes, and *BoolFalse* if it is aborted by the radiation monitor.

NOTE: This function is currently unused, and may disappear.

Concurrency: Synchronous

30.6.9 updateCntlSettings()

Protected member of: **SystemConfiguration**

Return Class: **void**

Documentation:

This function updates any settings which have changed in the DEA Interface controller.

Concurrency: Synchronous

30.6.10 updateDeaPower()

Protected member of: **SystemConfiguration**

Return Class: **Boolean**

Documentation:

This function tests DEA power, first shutting down all boards which should be turned off, and then powering on boards which should be powered on. This function returns *BoolTrue* if successful, and *BoolFalse* if it was aborted by the radiation monitor.

Concurrency: Synchronous

30.6.11 updateFepPower()

Protected member of: **SystemConfiguration**

Return Class: **Boolean**

Documentation:

This function updates the power settings of the FEPs. It first powers down all FEPs which should be off, and then powers on and runs the default FEP program on all FEPs which should be powered on. This function currently always returns *BoolTrue*.

Concurrency: Synchronous

31.0 DEA Housekeeper (36-53221 02)

31.1 Purpose

The DEA Housekeeper function provides the capability to acquire the contents of a specific set of DEA registers at a prescribed interval rate.

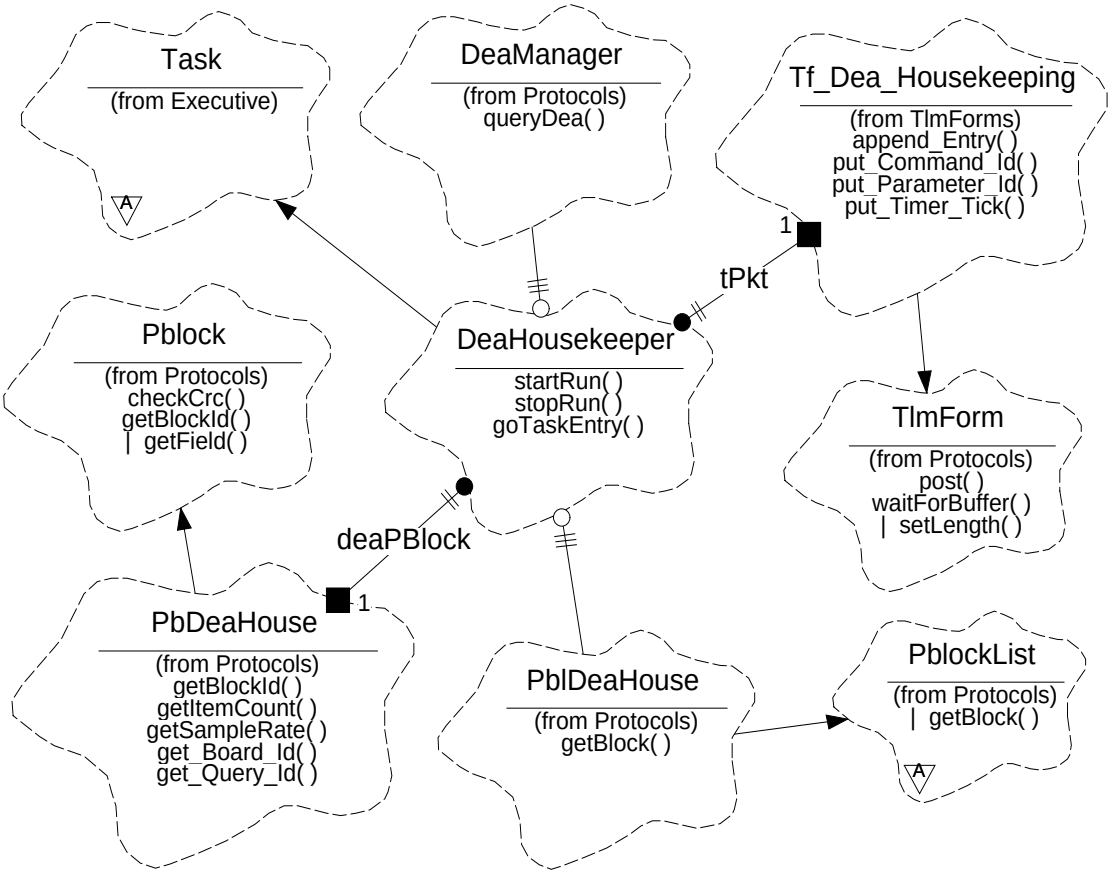
31.2 Uses

Use 1:: Start DEA Housekeeping
Use 2:: Acquiring the selected DEA register values
Use 3:: Stop DEA Housekeeping

31.3 Organization

Figure 148 illustrates the relationship between the classes used by the DEA Housekeeper.

FIGURE 148. DEA Housekeeping Class Relationships



The DEA Housekeeper uses the ***Executive*** and ***Protocols*** class categories.

DeaHousekeeper - This class is a subclass of ***Executive***::**Task**. It is responsible for accumulating and delivering DEA Housekeeping data as specified in the referenced parameter block, from the receipt of a start command until receipt of a stop command.

Tf_Dea_Housekeeping - This class encapsulates the representation of a DeaHousekeeper telemetry packet. It is a subclass of ***Protocols***::**TlmForm**.

Pblock - This class is responsible for manipulating the contents of a parameter block; extracting fields and verifying the block. It is a member of ***protocols*** class category.

PbDeaHouse - This class encapsulates the representation of a parameter block. It is a subclass of ***Protocols***::**Pblock**.

DeaManager - This class is responsible for interaction with the DEAs, when requested to acquire specified housekeeping data. It is a member of ***Protocols*** class category.

PblDeaHouse - This class is responsible for loading, maintaining, and delivering the sets of parameter blocks. It is a subclass of ***Protocols***::**pBlockList**.

TaskMonitor - This class (not shown) is a subclass of ***Executive***::**Task**. It interrogates each thread in turn, verifying that it is functioning. Failure to respond will cause a watchdog reset.

`notify` and `waitForEvent` and `requestEvent` - These are functions of ***Executive***::**Task** inherited by the DeaHousekeeper. They provide the connectivity between the clients request via the housekeeper public functions and the threads main process.

SystemClock - This class (not shown) provides the BEP tick count which is included in the telemetry Packet. It is a subclass of ***Executive***.

The entire command which loads the DEA Housekeeping Parameter Block will be stored into the block.

The block will contain: a command packet header, a DEA housekeeping header, and a set of target identifiers. The command packet header consisting of: message length, packet identifier, and an op-Code. The DEA housekeeping header contains, the slot index, the parameter block identifier, the block CRC, and the data sampling rate. This is followed by a series of one word target identifiers which indicate the board and the register to interrogate for housekeeping data.

31.4 Scenario

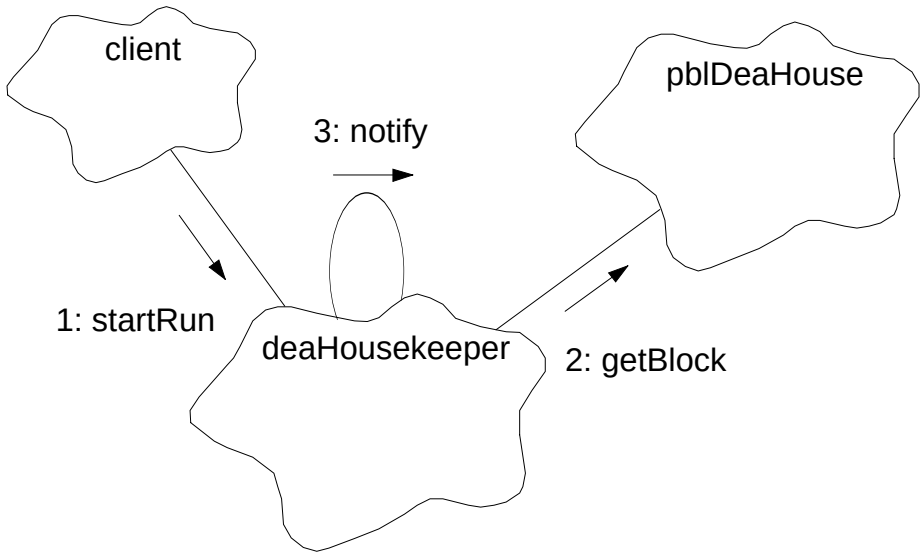
The `notify` function communicates the client's request to the DEA Housekeeper thread. There are three states to control the next action; START, SAMPLE, and STOP. The start request will obtain a verified parameter block, set *`nextAct`* to START and `notify` the thread. When the idle thread receives the notification it will establish housekeeping according to the block delivered, set the state to SAMPLE, and proceed, continuing housekeeping, until notified of a change in state. If currently sampling, it will restart. A stop request will verify the current block, set the state to STOP, and notify the thread.

31.4.1 Use 1:: Start DEA Housekeeping

Figure 149 illustrates the initiation of the DEA housekeeping process.

1. To begin a DEA housekeeping run, the client must provide the slot index from which the parameter block will be extracted for the run, and the command Id from the packet header which is to be echoed in the telemetry identifying the instigating agent for the action. Then it must engage this tasks public function `startRun` to start DEA housekeeping.

FIGURE 149. Start DEA Housekeeping



2. When a client requests that a run be started, an instance of **`PBlock`** is created. `startRun` will acquire a copy of the designated block using **`PblDeaHouse`**::`getBlock`. After copying the block, `getBlock` performs a CRC check on the copy, returning BoolFalse if it can not deliver a valid copy. If a block is not delivered, `startRun` returns CMDRESULT_BAD_LOAD to the client and the request is not forwarded. The **`PBlock`** instance holds the copy.
3. The current state of the thread is checked. Should the thread be executing a prior run request, a missed stop command is assumed. In that case the START state will cause a termination of the current run and initiation of the requested run. `startRun` will set the return argument to CMDRESULT_ALREADY. In this event, the last data-point in

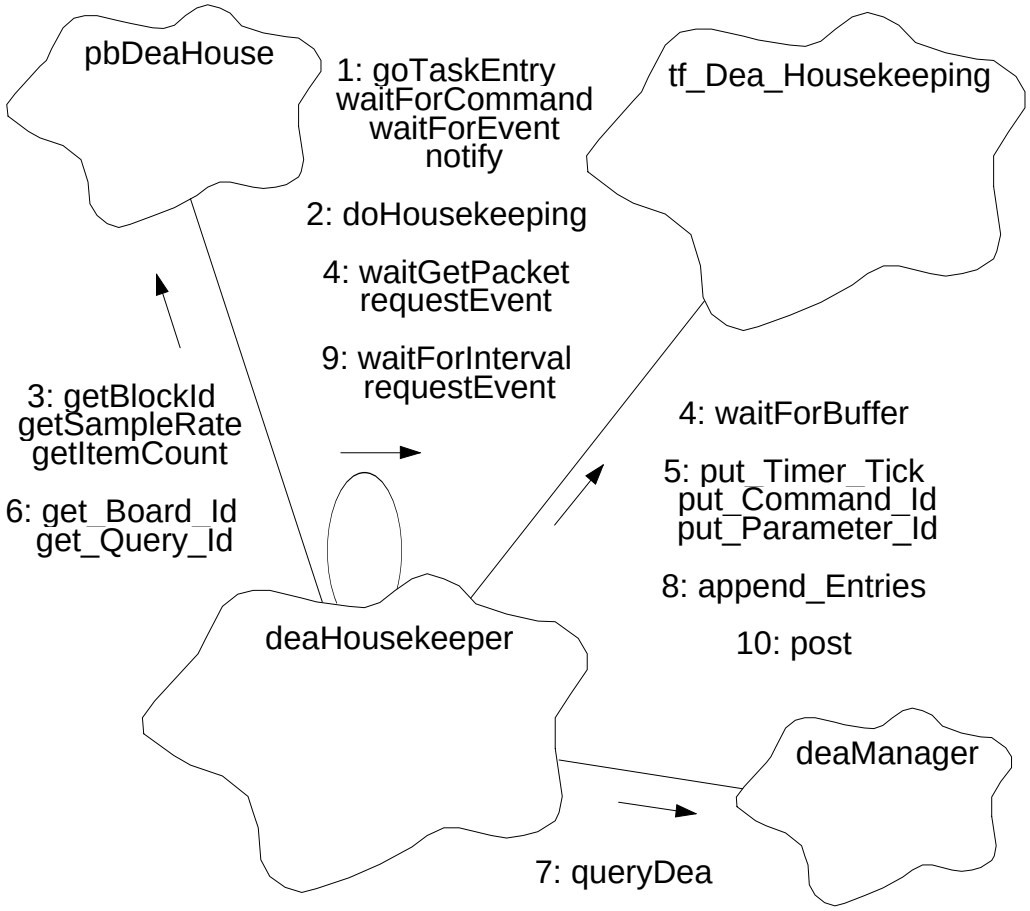
the current packet may reference the new parameter block. The client has the responsibility to convey this possibility. If the current state of the thread is stopped, the process will set the *nextAct* to START and set the return argument to CMDRESULT_OK. In either case, the startRun function will send a notify request to the thread before returning.

31.4.2 Use 2:: Acquiring DEA Housekeeping Data

Figure 150 illustrates the acquisition of the DEA housekeeping process.

1. The main thread of this task, goTaskEntry, is initiated during BEP start-up. It consists of a forever loop in which the task idles in waitForCommand using the inherited waitForEvent while waiting to call **TaskMonitor**::respond to answer the **TaskMonitor**::query (neither shown) or to the notify sent by the startRun or stopRun public functions indicating that a request has been received and awaits proper action. Receipt of a startRun request causes the task to enter a loop in which it calls (or recalls) doHousekeeping whenever the *START* state is set (or reset).
2. In doHousekeeping, the process sets *nextAct* to SAMPLE.
3. The DEA header values, block Id and sample rate, are extracted from the parameter block using getBlockId and getSampleRate, and the item count is returned by getItemCount.
4. It then enters a loop which it exits only when a new notify request is detected. It creates an instance of the DEA telemetry packet, **Tf_Dea_Housekeeping**. Then it enters waitGetPacket to obtain a packet buffer using the packet instance's waitForBuffer which it inherits through **TlmForm**. waitGetPacket. It will use requestEvent to check for a notify message. It will respond to a **taskMonitor**::query (not shown), and will return *BoolFalse* (no packet buffer) if notified of a new request from the **DeaHousekeeper** public functions startRun or stopRun. As the result of a new request, doHousekeeping will return to goTaskEntry. Obtaining the packet buffer, waitGetPacket returns *BoolTrue*.
5. Getting the time reference from **systemClock**::currentTime (not shown), doHousekeeping will install the time and the command and block Ids into the DEA Header of the packet using functions provided by the packet instance. They are: put_timer_Tick, put_Command_Id, and put_Parameter_Id.
6. The function drops into a loop in which it obtains the target indicators; the board index, and the register index. They are extracted from the parameter block using get_Board_Id, and get_Query_Id respectively.
7. It uses the **deaManager**::queryDea to obtain the targeted register value, providing a default value if there is no response. The loop is cycled for *itemCount* times.

FIGURE 150. Acquisition of Housekeeping Data



8. It then uses the packet instance function `append_Entries` to load the target Identifiers, *`boardIndex`* and *`queryIndex`*, and the register response *`value`* into the packet.
9. The process then tests the state, exiting the loop if it is not still SAMPLE, else it uses `waitForInterval` to check for notifications from the **`TaskMonitor`** or from this tasks public functions using the inherited function `requestEvent`. It then waits the requisite interval, *`sampleRate`*, before processing the next data point.
10. Having completed the data set, or having been told to stop (or re-start), thus truncating the data set, **`Tf_Dea_Housekeeping`** will forward the packet using the inherited function `post`.

31.4.3 Use 3:: Stop DEA Housekeeping

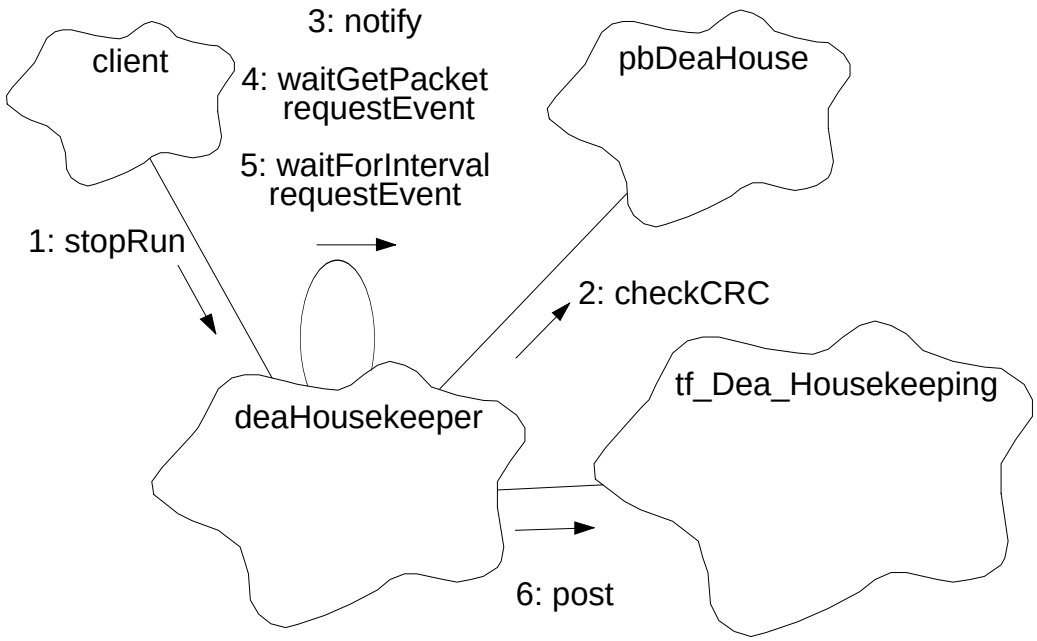
Figure 151 illustrates the termination of the DEA housekeeping process.

1. The stop run request is handled by the `stopRun` public function called by the client. It does not require arguments.
2. It will use `checkCRC` to verify the current parameter block returning

CMDRESULT_BAD_CRC on failure. If the thread is not processing data, the interface assumes that it is stopped and will return CMDRESULT_ALREADY, else it will return CMDRESULT_OK.

3. Regardless of the outcome of these checks, the interface will `notify` the thread to stop the run. If `goTaskEntry` is waiting for a command, it will receive the notification from the `stopRun` function. The stop run request is ignored by `goTaskEntry`, since, if one is received, the process is already stopped, and no action is required.

FIGURE 151. Stop DEA Housekeeping



4. If housekeeping data is being collected, `doHousekeeping` will receive the request, and act upon it in the course of its processing. While obtaining a packet buffer, it uses `waitGetPacket` which uses `requestEvent` to check pending notifications from **`TaskMonitor`**::`query` and from the public functions `startRun` and `stopRun`. Receiving a notification from either public function will cause a return without a buffer. Without a packet to finalize, `dohousekeeping` will return.
5. `doHousekeeping` will also respond to notifications after the acquisition of each data value using `waitForInterval`, which uses `requestEvent` to check pending notifications as above. Receiving a notification from either public function will cause a return without delaying for the *`sampleRate`* interval.
6. `doHousekeeping` will terminate the run and deliver the packet using `post`.

31.5 Class DeaHousekeeper

Documentation:

The DEA Housekeeper is responsible for modulated acquisition and periodic telemetry of engineering information from the DEA subsystem. It is started and stopped by command.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Public Uses:

`PblDeaHouse`

Implementation Uses:

`DeaManager`

Public Interface:

Operations:
```
DeaHousekeeper()
goTaskEntry()
startRun()
stopRun()
```

Protected Interface:

Has-A Relationships:

`unsigned` *`cmdGndId`*: The value the ground uses to identify a command. It is echoed in packets generated in response to the command.

`unsigned` *`sampleRate`*: Initially holds the number of seconds which should elapse between requests for housekeeping data. It is multiplied by TICKS_PER_SECOND to be used by `sleep`().

Operations: `doHousekeeping()`
`waitForCommand()`
`waitForInterval()`
`waitGetPacket()`

Private Interface:

Has-A Relationships:

`PbDeaHouse` *`deaPBlock:`* An instance of a DEA parameter block.

`KeeperAction` *`nextAct`*: This instance variable conveys the action requested by the client, from the public binding function to the housekeeper. It will reflect the pending state of the housekeeper; START, STOP, or SAMPLE (continue).

`Tf_Dea_Housekeeping` *`tPkt`*: An instance of a DEA telemetry packet.

Concurrency: Active

Persistence: Persistent

31.5.1 DeaHousekeeper()

Public member of: **`DeaHousekeeper`**

Arguments:

`unsigned` *`taskId`*

Documentation:

The identity of this task is contained in *`taskId`*. This constructor initializes instance variables.

Concurrency: Sequential

31.5.2 goTaskEntry()

Public member of: **`DeaHousekeeper`**

Return Class: **`void`**

Documentation:

This function is the main entry point of the task.

Semantics:

This thread is idle unless commanded to start data acquisition according to parameters stored in a designated block. A start run notification will initiate housekeeping by calling `doHousekeeping`. Housekeeping data will continue to be obtained until the housekeeper is commanded to stop. A stop notification received by this function is ignored: because it is stopped.

Concurrency: Synchronous

31.5.3 startRun()

Public member of: **DeaHousekeeper**

Return Class: **CmdResult**

Arguments:

unsigned *slotIndex*
unsigned *cmdGndId*

Documentation:

This function acquires and verifies the designated block (CRC check) and instructs the DEA Housekeeper task to start acquiring and sending information from the DEA. *slotIndex* is the index (acis location) of the DEA Housekeeping parameter block to use for the run. *cmdGndId* is the ground designation of the command sent to instigate this action. It is included in response telemetry

Semantics:

This function will create an instance of a parameter block then extract a copy of the designated block using **pblDeaHouse**::getBlock. A faulty attempt to acquire the block, or a bad CRC check of the block, will cause a return with state CMDRESULT_BAD_LOAD. The request is not forwarded. With a valid parameter block, the threads state is tested, if it is running, the return state is set to CMDRESULT_ALREADY, but if it is stopped, the return state set to CMDRESULT_OK. The pending-action state variable, *nextAct*, is set to START. The function will use notify to inform the thread of the request before returning.

Concurrency: Synchronous

31.5.4 stopRun()

Public member of: **DeaHousekeeper**

Return Class: **CmdResult**

Documentation:

This function instructs the DEA housekeeper to stop acquiring and sending information from the DEA. The last data obtained will be posted.

Semantics:

This function will check the current block CRC using **deaPBlock**::checkCrc and set the return state CMDRESULT_BAD_CRC if it is invalid. It will set the state CMDRESULT_ALREADY if processing state is stopped, otherwise it will set the state to CMDRESULT_OK. The pending-action state variable is set to STOP. The thread will be informed using notify.

Concurrency: Synchronous

31.5.5 doHousekeeping()

Protected member of: **DeaHousekeeper**

Return Class: **void**

Documentation:

This function periodically acquires and telemeters DEA housekeeping values. The parameter block to use was loaded by the public *startRun* function. This function runs until commanded to stop.

Semantics:

The pending-action state is set to SAMPLE. The parameter block entries for the science identifier, the count of housekeeping items, and the rate at which housekeeping data is acquired are obtained using `getBlockId`, `getItemCount`, and `getSampleRate`. The main process loop is entered and will be cycled until a stop run directive is detected. A **`Tf_Dea_Housekeeping`** packet instance is passed to `waitGetPacket` to be associated with a packet buffer. It also will check for new commands and for queries from the TaskMonitor. If a new request is detected, the loop will be terminated and this function will return. The packet functions will be used to load data into the packet. `currentTime` is used to obtain the BEP tick count. The tick count, the command identity which was received from the client, and the science block identity are installed in the packet using `put_Timer_Tick`, `put_Command_Id`, and `put_Parameter_Id`. A housekeeping data loop is entered during which the target board and register indices will be extracted from the block using `get_Board_Id` and `get_Query_Id`. Then a response value from the target requested from **`deaManager`**::`queryDea` or a default value will be installed in the packet using `append_Entries`. `waitForInterval` is used to check for new commands and for queries from the **`TaskMonitor`**. If a new request is detected, the loop will be terminated. When the housekeeping requests have been fulfilled, or the run terminated with a partially filled packet, this function will access `post` to forwarded the packet for transmission.

Concurrency: Guarded

31.5.6 waitForCommand()

Protected member of: **DeaHousekeeper**

Return Class: **void**

Documentation:

This function waits until the task is commanded to start (or to stop) acquiring housekeeping values while responding to **TaskMonitor**::query.

Semantics:

This function consists of a loop in which it waits; using waitForEvent for notification of a **TaskMonitor**::query which initiates **TaskMonitor**::respond, or a request from one of its public functions which allows it to leave the loop and return.

. Concurrency: Guarded

31.5.7 waitForInterval()

Protected member of: **DeaHousekeeper**

Return Class: **void**

Documentation:

This function waits for *sampleRate* seconds. It is used to space out engineering value acquisitions while responding to queries from the **TaskMonitor** or for a command from the housekeeping public functions.

Semantics:

This function uses *requestEvent* to determine if the **TaskMonitor** has interrogated this process or if a public function has sent a notify to this process of a received command. If so, it uses **TaskMonitor**::respond to answer the **TaskMonitor** or returns so its caller may address the command. With no pending directive, it will sleep for the designated interval, and return.

Concurrency: Guarded

31.5.8 waitGetPacket()

Private member of: **DeaHousekeeper**

Return Class: **Boolean**

Arguments:

TlmForm & *pkt*

Documentation:

This function waits until a packet buffer becomes available while checking for **TaskMonitor** or **DeaHousekeeper** notifications. The buffer, in due course, will contain the data of the packet instance *pkt*.

Semantics:

This function checks for the **TaskMonitor**::query or the **DeaHousekeeper**::startRun or **DeaHousekeeper**::stopRun notifications. It will respond to queries, and return BoolFalse (no packet buffer) if a new housekeeping request is received. It then waits for a fresh packet buffer for a timed duration using waitForBuffer. On failure to obtain a packet in that time, it loops to check and to wait again. When a packet buffer is obtained, it will return BoolTrue.

Concurrency: Guarded

32.0 Bad Pixel and Column Map Classes (36-53240 A)

32.1 Purpose

The purpose of the Bad Pixel and Column Map classes are to maintain a persistent list of bad CCD pixels and columns within the instrument. Within the instrument, there is one bad pixel map, and two bad column maps. The bad column maps are respectively associated with Timed Exposure mode and Continuous Clocking mode. These lists are used by science processing to prevent damaged pixels from saturating telemetry with non-X-ray event data.

32.2 Uses

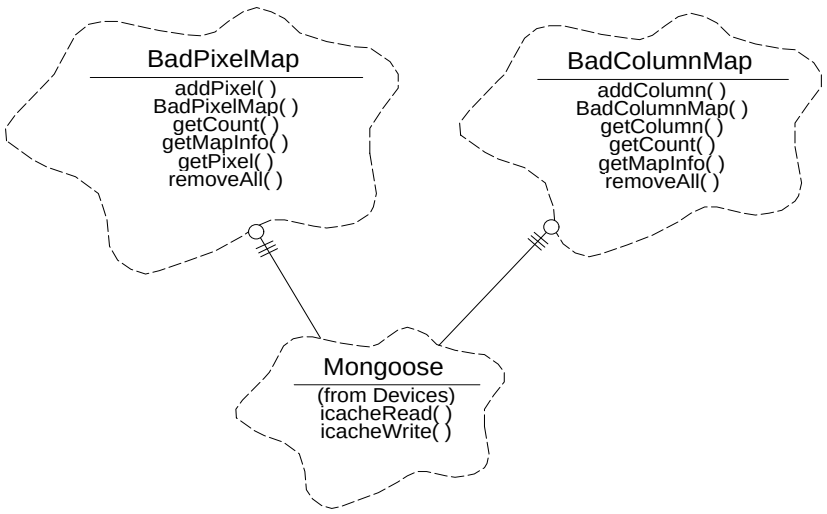
The following lists the use of the Bad Pixel and Column Map classes:

Use 1:: Append a bad pixel entry to the end of a bad pixel or column map
Use 2:: Remove all entries from a bad pixel or column map
Use 3:: Retrieve an entry from a bad pixel or column map
Use 4:: Retrieve the address and length of a map (for telemetry dump purposes)

32.3 Organization

Figure 152 illustrates the class relationships used by the Bad Pixel and Column Map classes. These classes include the **`BadPixelMap`** class, which is responsible for maintaining a list of bad pixels, and a **`BadColumnMap`** class, which is responsible for maintaining a list of bad column entries. In order to prevent stray pointers from corrupting the tables, their contents are maintained within Instruction Cache RAM (I-cache). The **`BadPixelMap`** and **`BadColumnMap`** classes both use the **`Mongoose`** class to read and write to this RAM.

FIGURE 152. Bad Pixel and Column Map Class Relationships



`BadPixelMap`- This class represents a list of bad pixels for all CCDs in the system. This class maintains the list in I-cache and provides functions to append new entries to the end of the list (`addPixel`), remove all entries from the list (`removeAll`), retrieve an entry from the list (`getPixel`) and get the total number of entries currently stored in the list (`getCount`). In order to support dumping the list to telemetry, the class also provides a function which returns the address of the start of the list, and the number of 32-bit words contained in the list (`getMapInfo`).

`BadColumnMap`- This class represents a list of bad columns for all CCDs in the system. This class maintains the list in I-cache and provides functions to append new entries to the end of the list (`addColumn`), remove all entries from the list (`removeAll`), retrieve an entry from the list (`getColumn`) and get the total number of entries currently stored in the list (`getCount`). In order to support dumping the list to telemetry, the class also provides a function which returns the address of the start of the list, and the number of 32-bit words contained in the list (`getMapInfo`).

`Mongoose`- This class is provided by the ***`Devices`*** class category, and is used by the **`BadPixelMap`** and **`BadColumnMap`** classes to write and read data to and from I-cache RAM (`icacheWrite`, `icacheRead`).

32.4 Memory Layouts

32.4.1 I-cache Memory Map

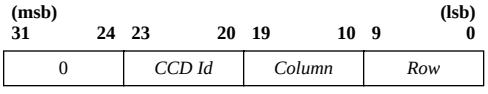
In order to reduce the opportunity that writes through a corrupted data pointer will corrupt the bad pixel and column maps, these maps are maintained within Instruction Cache RAM. Each map consists of a 32-bit entry count, followed by zero or more entries. For bad pixel maps, each entry is 32-bits wide. For bad column maps, each entry is 16-bits wide. Table 27 illustrates a proposed layout for the bad pixel and column maps.

TABLE 27. I-cache Bad Pixel and Column Map Layout (TBD)

Region		Address	Byte Size	Description
Patch Area		0x800fffff 0x800d7c00	0x30400	
Bad Pixel Map	Entries	0x800cdc04	0x9ffc	Array of bad pixel map entries (max. 10239)
	Count	0x800cdc00	0x4	Number of 32-bit bad pixel map entries in the table
Continuous Clocking Bad Column Map	Entries	0x800cc404	0x17fc	Array of bad column map entries (max. 3070 entries)
	Count	0x800cc400	0x4	Number of 16-bit bad column map entries in the table
Timed Exposure Bad Column Map	Entries	0x800cac04	0x17fc	Array of bad column map entries (max. 3070 entries)
	Count	0x800cac00	0x4	Number of 16 bit bad column map entries in the table
Compression Tables		0x800c2c00	0x8000	
Parameter Blocks		0x800c0400	0x2800	
System Configuration		0x800c0000	0x400	
Code		0x80080000	0x40000	

32.4.2 Bad Pixel Entry Format

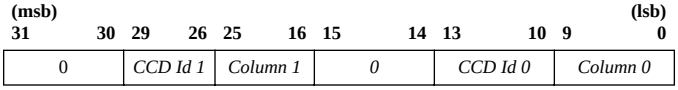
The bit layout for a bad pixel entry is as follows:



Where *row* and *column* specify the row and column position of the pixel within the CCD, and *CCD Id* specifies which CCD has the bad pixel. 0 indicates that the corresponding bits are set to 0 in a map entry.

32.4.3 Bad Column Entry Format

Bad column entries are bit-packed into each 32-bit entry location within I-cache, starting with the lower 16-bits. If there are an odd number of bad column entries in the table, the last 32-bit word in the table will contain 0 in its upper 16-bits. (NOTE: There is no ambiguity since he count at the beginning of the table reflects the number of column entries, not pairs of entries). The format of a 32-bit word containing two entries is as follows:



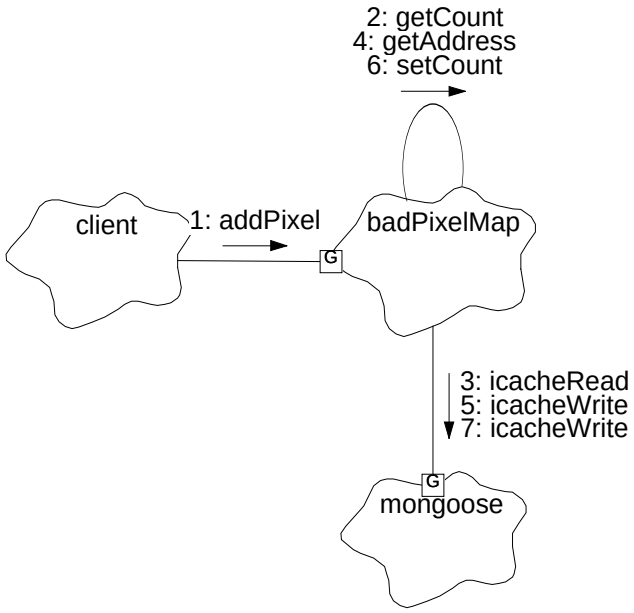
Where *Column* specifies the position of the column pixel within the CCD, and *CCD Id* specifies which CCD has the bad pixel. 0 indicates that the corresponding bits are set to 0 in a map entry.

32.5 Scenarios

32.5.1 Use 1: Append a bad pixel entry to the end of a map

Figure 153 illustrates the steps used by the client to append a bad pixel to the end of the bad pixel map. The steps to append a bad column are similar.

FIGURE 153. Append Bad Pixel to map

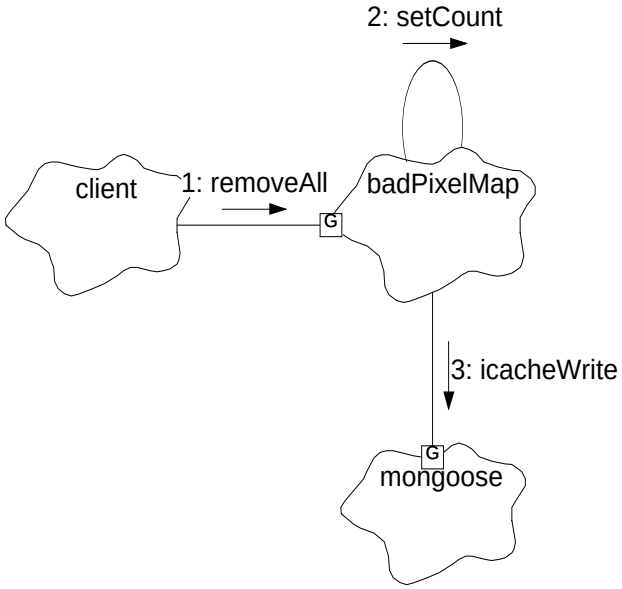


1. The *client* appends a pixel to the end of the bad pixel list by passing the pixel's CCD identifier, and row and column identifiers to *badPixelMap*.addPixel().
2. addPixel() gets the current end of the map using getCount().
3. getCount() uses *mongoose*.icacheRead() to read the entry count from within I-cache.
4. addPixel() checks to see if the table is full, and if so, returns the condition to the client. If the table is not full, addPixel() then retrieves the address of the entry slot just after the last entry using getAddress().
5. addPixel() then forms the entry value and write it to I-cache using *mongoose*.icacheWrite(). For the Bad Column map, if the entry slot index is odd, addColumn() (not illustrated) uses *mongoose*.icacheRead() to read the current 32-bit word from RAM, places the new entry value into the upper 16-bits of the word, and writes the value back out using *mongoose*.icacheWrite().
6. addPixel() then increments the current entry count and stores the value using setCount().
7. setCount() writes the new entry counter to I-cache using *mongoose*.icacheWrite().

32.5.2 Use 2: Remove all entries from a map

Figure 154 illustrates the steps used by the client to remove all entries from the bad pixel map. The steps to remove all entries from a bad column map are similar.

FIGURE 154. Delete contents of map

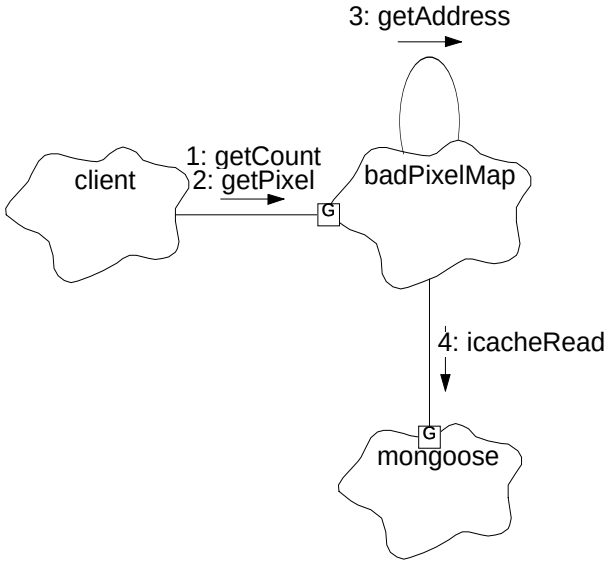


1. The *client* empties the bad pixel map by calling *badPixelMap*.removeAll().
2. removeAll() passes zero to setCount() to indicate the map is now empty.
3. setCount() stores the passed entry count into I-cache using *mongoose*.icacheWrite().

32.5.3 Use 3: Retrieve an entry from a map

Figure 155 illustrates the steps used by the client to retrieve an entry from the bad pixel map. The steps to obtain a bad column entry are similar.

FIGURE 155. Get Bad Pixel entry

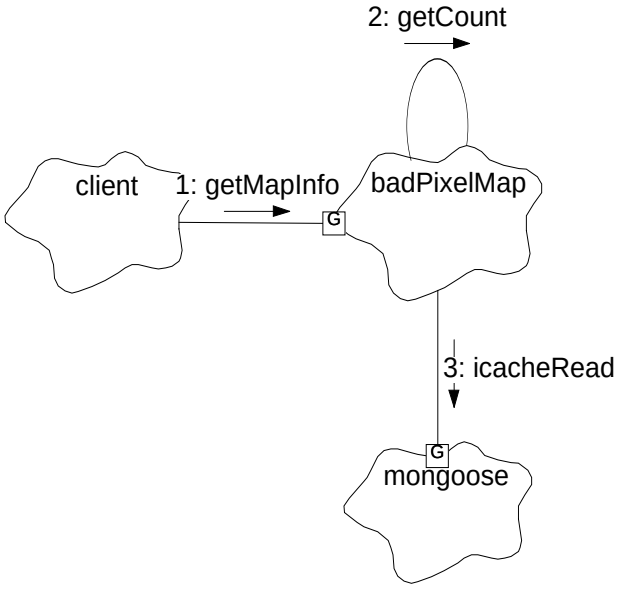


1. The client obtains the number of entries currently in the table using *`badPixelMap`*.`getCount`().
2. The client then iteratively calls *`badPixelMap`*.`getPixel`() to obtain the contents of each bad pixel map entry.
3. `getPixel`() uses `getAddress`() to obtain the memory location of the desired map entry.
4. `getPixel`() then reads the entry using *`mongoose`*.`icacheRead`(), and returns the entry's CCD identifier, and row and column position to the caller. For the Bad Column Map, if the requested index is odd, the function shifts the upper 16-bits of the read word into the lower 16-bits, and then extracts the entry's fields from the lower 16-bits.

32.5.4 Use 4: Retrieve the address and length of a map

Figure 156 illustrates the steps used by a client to obtain the address and word length of the bad pixel map. The steps to obtain the corresponding address and length of a bad column map are similar.

FIGURE 156. Get Map Address and Length



1. The client gets the I-cache address of the map, and the number of words in the map using *`badPixelMap`*.`getMapInfo`().
2. `getMapInfo`() retrieves the current number of entries using `getCount`().
3. `getCount`() reads the entry count value from I-cache using *`mongoose`*.`icacheRead`(). `getMapInfo`() then returns the map's address and word count to the caller.

32.6 Class BadPixelMap

Documentation:

This class represents the Bad Pixel Map within the instrument. It is responsible for managing the list of bad CCD pixels within I-cache.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:
```
BadPixelMap()
addPixel()
getCount()
getMapInfo()
getPixel()
removeAll()
```

Protected Interface:

Operations:
```
getAddress()
setCount()
```

Private Interface:

Has-A Relationships:

unsigned* const *baseAddress*: This is the base address of the bad pixel map in i-cache.

const unsigned *maxCount*: This is the maximum number of entries in the bad pixel map table.

Concurrency: Guarded

Persistence: Persistent

32.6.1 BadPixelMap()

Public member of: **BadPixelMap**

Arguments:

unsigned* *base*
unsigned *words*

Documentation:

This is the constructor for the bad pixel map. *base* is the base address of the map in I-cache, and *words* is the maximum number of words that can be stored in the map. The initialization statements for the constructor set *baseAddress* to the passed *base*, and *maxCount* to the passed *words*.

Concurrency: Sequential

32.6.2 addPixel()

Public member of: **BadPixelMap**

Return Class: **Boolean**

Arguments:

CcdId *ccdid*
unsigned *row*
unsigned *column*

Documentation:

This function adds a pixel to the bad pixel map. *ccdid* is the CCD which contains the bad pixel. *row* and *column* identify the pixel's row and column location within the CCD. If successful, the function returns *BoolTrue*. If the map is full and the pixel cannot be stored, it returns *BoolFalse*.

Semantics:

This forms the bit-field entry using the passed arguments. It then uses `getCount`() to get the number of entries currently in the map. If the result is less than *maxCount*, the function uses `getAddress`() to get the virtual address of the last written entry, and uses *mongoose*.`icacheWrite`() to write the new entry onto the end of the map. It then uses `setCount`() to store the incremented entry count into I-cache.

Concurrency: Guarded

32.6.3 getAddress()

Protected member of: **BadPixelMap**

Return Class: **unsigned***

Arguments:

unsigned *index*

Documentation:

This function obtains the I-cache address for the table entry indicated by *index*. The function uses `getCount`() to obtain the number of entries in the table. If *index* is beyond the end of the table, this function returns 0, otherwise, it adds *index* + 1 to *baseAddress* (the plus 1 handles the count field at the beginning of the table) to form the address of the entry, and returns the computed table entry address.

Concurrency: Guarded

32.6.4 getCount()

Public member of: **BadPixelMap**

Return Class: **unsigned**

Documentation:

This function returns the number of bad pixel entries currently in the bad pixel map. This function uses *mongoose*.`icacheRead`() to read the count field from the map.

Concurrency: Guarded

32.6.5 getMapInfo()

Public member of: **`BadPixelMap`**

Return Class: **`void`**

Arguments:

`const unsigned*&` *`addr`*
`unsigned&` *`wordcnt`*

Documentation:

This function retrieves the base address of the bad pixel map, and the number of 32-bit words currently stored in the map. On return, *`addr`* contains the address of the map in I-cache (*`baseAddress`*), and *`wordcnt`* contains the number of words in the map (result of `getCount`() + 1).

Concurrency: Guarded

32.6.6 getPixel()

Public member of: **BadPixelMap**

Return Class: **Boolean**

Arguments:

unsigned *index*
CcdId& *ccdout*
unsigned& *rowout*
unsigned& *colout*

Documentation:

This function retrieves the bad pixel indexed by *index*. On return, *ccdout* contains the CCD which contains the bad pixel, and *rowout* and *colout* contain the row and column position within the CCD. If *index* is within range, the function returns *BoolTrue*. If *index* is beyond the last entry in the map, the function returns *BoolFalse*.

Semantics:

This starts by getting the address corresponding to the entry indicated by index using getAddress(). If the entry is valid (i.e. result is not zero), the function uses *mongoose*.icacheRead() to read the entry, and then extracts the bit-fields from the entry and stores the values into *ccdout*, *rowout* and *colout*.

Concurrency: Guarded

32.6.7 removeAll()

Public member of: **BadPixelMap**

Return Class: **void**

Documentation:

This function removes all pixels from the Bad Pixel Map by writing 0 to the entry count at the start of the map via setCount().

Concurrency: Guarded

32.6.8 setCount()

Protected member of: **`BadPixelMap`**

Return Class: **`void`**

Arguments:

`unsigned` *`count`*

Documentation:

This function sets the number of pixels currently stored in the bad pixel map. This function calls *`mongoose`*`.icacheWrite()` to store count into the first location of the map.

Concurrency: Guarded

32.7 Class BadColumnMap

Documentation:

This class represents the Bad Column Map within the instrument. It is responsible for managing the list of bad CCD columns within I-cache.

Export Control: Public

Cardinality: 2

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:
`BadColumnMap()`
`addColumn()`
`getColumn()`
`getCount()`
`getMapInfo()`
`removeAll()`

Protected Interface:

Operations:
`getAddress()`
`setCount()`

Private Interface:

Has-A Relationships:

`unsigned* const` *`baseAddress`*: This is the base address of the bad column map in i-cache.

`const unsigned` *`maxCount`*: This is the maximum number of entries in the bad column map table.

Concurrency: Guarded

Persistence: Persistent

32.7.1 BadColumnMap()

Public member of: **BadColumnMap**

Arguments:

unsigned* *base*
unsigned *words*

Documentation:

This is the constructor for the bad column map. *base* is the base address of the map in I-cache, and *words* is the maximum number of words that can be stored in the map. The initialization statements for the constructor set *baseAddress* to the passed *base*, and *maxCount* to the passed *words*.

Concurrency: Sequential

32.7.2 addColumn()

Public member of: **BadColumnMap**

Return Class: **Boolean**

Arguments:

CcdId *ccdid*
unsigned *column*

Documentation:

This function adds a column to the bad column map. *ccdid* is the CCD which contains the bad column. *column* identifies the column's location within the CCD. If successful, the function returns *BoolTrue*. If the map is full and the column cannot be stored, it returns *BoolFalse*.

Semantics:

This forms a right-justified bit-field entry using the passed arguments. It then uses getCount() to get the number of entries currently in the map. If the result is less than *maxCount*, the function uses getAddress() to get the virtual address of the last written entry. If the result is odd, it then uses mongoose.icacheRead() to read the low order bits of the last entry, shifts the new entry into the upper 16-bits and combines the two. Otherwise, if the entry count is even, the function increments the address pointer to point to the next 32-bit word in the table. If then uses *mongoose*.icacheWrite() to write the new entry onto the end of the map. It then uses setCount() to store the incremented entry count into I-cache.

Concurrency: Guarded

32.7.3 getAddress()

Protected member of: **`BadColumnMap`**

Return Class: **`unsigned*`**

Arguments:

`unsigned` *`index`*

Documentation:

This function obtains the I-cache address for the table entry indicated by *`index`*. The function uses `getCount`() to obtain the number of entries in the table. If *`index`* is beyond the end of the table, this function returns 0, otherwise, it adds (*`index`*/2) + 1 to *`baseAddress`* (NOTE: The divide by two is because there are two entries per 32-bit I-cache word, and the plus 1 handles the count field at the beginning of the table) to form the address of the entry, and returns the computed table entry address.

Concurrency: Guarded

32.7.4 getColumn()

Public member of: **BadColumnMap**

Return Class: **Boolean**

Arguments:

unsigned *index*
CcdId& *ccdout*
unsigned& *colout*

Documentation:

This function retrieves the bad column indexed by *index*. On return, *ccdout* contains the CCD which contains the bad column, and *colout* contains the column position within the CCD. If *index* is within range, the function returns *BoolTrue*. If *index* is beyond the last entry in the map, the function returns *BoolFalse*.

Semantics:

This starts by getting the address corresponding to the entry indicated by index using getAddress(). If the entry is valid (i.e. result is not zero), the function uses *mongoose*.icacheRead() to read word containing the entry. If *index* is odd, it then shifts the read word to put the desired entry into the lower 16-bits of the word. It then extracts the bit-fields from the entry in the lower 16-bits and stores the values into *ccdout*, *rowout* and *colout*.

Concurrency: Guarded

32.7.5 getCount()

Public member of: **BadColumnMap**

Return Class: **unsigned**

Documentation:

This function returns the number of bad column entries currently in the bad column map. This function uses *mongoose*.icacheRead() to read the count field from the map.

Concurrency: Guarded

32.7.6 getMapInfo()

Public member of: **BadColumnMap**

Return Class: **void**

Arguments:

const unsigned*& *addr*
unsigned& *wordcnt*

Documentation:

This function retrieves the base address of the bad column map, and the number of 32-bit words currently stored in the map. On return, *addr* contains the address of the map in I-cache (*baseAddress*), and *wordcnt* contains the number of words in the map. The *wordcnt* is computed by dividing the result of getCount() by two, rounding up to the nearest word, and adding 1 (for the count field at the start of the table).

Concurrency: Guarded

32.7.7 removeAll()

Public member of: **`BadColumnMap`**

Return Class: **`void`**

Documentation:

This function removes all columns from the Bad Column Map by writing 0 to the entry count at the start of the map via `setCount`().

Concurrency: Guarded

32.7.8 setCount()

Protected member of: **`BadColumnMap`**

Return Class: **`void`**

Arguments:

`unsigned` *`count`*

Documentation:

This function sets the number of columns, *`count`*, currently stored in the bad column map using *`mongoose`*.`icacheWrite`().

Concurrency: Guarded

33.0 Science Management Classes (36-53222 B)

33.1 Purpose

The purpose of the Science Management Classes are to implement the top-level control of science data processing on the Back End Processor. This section discuses the overall control of science operations, and discuss the **`ScienceManager`**, **`ScienceMode`**, and **`ProcessMode`** classes.

Specific science modes (**`SmTimedExposure`**, **`SmContClocking`**), science data processing and the relevant classes are described in Section 37.0.

33.2 Uses

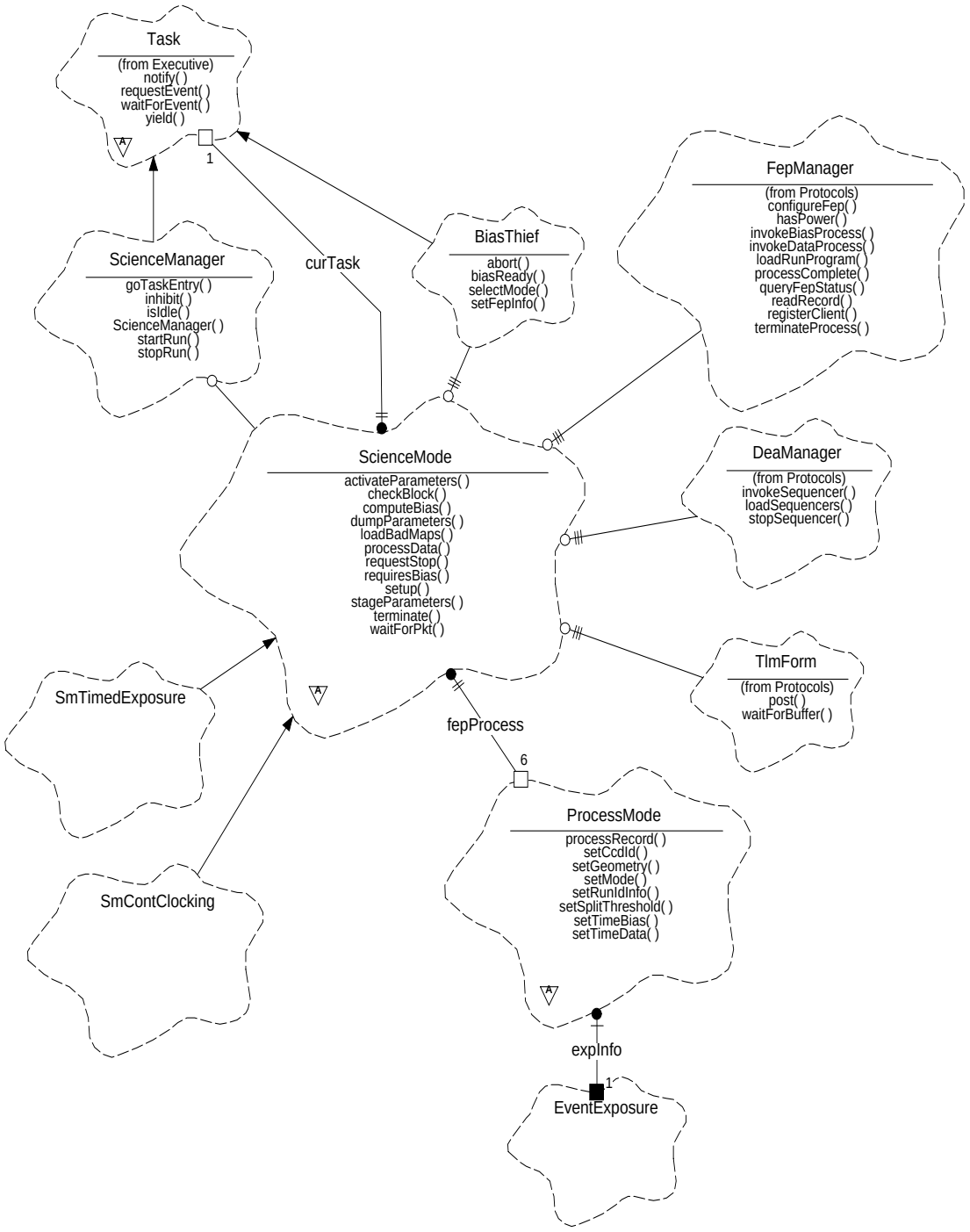
The following lists the uses of the Science Management classes. In practice, the first use listed covers each of the subsequent uses, which are individually listed to indicate each of the main steps needed to perform a science or bias run:

Use 1:: Perform a bias or science run
Use 2:: Initiate the setup of the hardware and software for a science or bias run
Use 3:: Initiate the dump of parameters used by a science or bias run
Use 4:: Initiate a bias computation for each of the selected CCDs
Use 5:: Initiate image acquisition and processing for each of the selected CCDs

33.3 Organization

Figure 157 illustrates the overall relationships between the Science Manager and Science Mode classes.

FIGURE 157. Science Management Classes



`ScienceManager` - This class is a subclass of ***`Executive`***::**`Task`** and is responsible for handling requests to start (`startRun`) and stop (`stopRun`) science and/or bias runs, and for directing a science mode through each stage of commanded run (`goTaskEntry`). This class has two types of functions, binding-functions and implementation functions. Binding functions are those functions which may be called from other tasks, to direct the science manager to take some sort of action (`startRun`, `stopRun`, `notify`,

inhibit), or to query the state of the task (isIdle). Implementation functions are used internally by the science task, and are typically called by the task itself (requestEvent, waitForEvent, yield). The **ScienceManager** uses a passed **ScienceMode** to implement each stage of a bias or science run.

ScienceMode - This abstract class is responsible for providing top level functions for each stage of a science bias or data processing run. These functions include checking the integrity of a parameter block (checkBlock), setting up for the run (stageParameters, activateParameters, setup), dumping the parameters for the run (dumpParameters), determining if a bias computation is required by the mode's parameters (requiresBias), performing a bias computation (computeBias), loading bad pixel and column maps (loadBadMaps), processing science data (processData), stopping data processing (requestStop), and shutting down at the end of a run (terminate). It also provides a function used by the data processing classes, **ProcessMode**, to wait for telemetry packets (waitForPkt). Each instance of **ScienceMode** contains an array of pointers to **ProcessMode** instances, one for each Front End Processor in the instrument. Each referenced **ProcessMode** is used by the **ScienceMode** to handle science data records produced by a single Front End Processor. Not shown in the figure is the collection of utility member functions, used by the **ScienceMode** class, and its child classes, to implement the details of the run. The use of these functions is shown in Section 33.5, and are described in detail in Section 37.0.

SmTimedExposure - This is a subclass of **ScienceMode** and is responsible for configuring and performing a Timed Exposure bias or data processing run. This class is described in detail in Section 37.0. When configuring the **ProcessMode** pointers, this class uses subclasses of **ProcessMode** to handle different types of Timed Exposure data processing modes. For raw telemetry mode, it uses instances of the **PmTeRaw** class (not shown). For histogram mode, it uses **PmHist** instances (not shown). When processing events, **SmTimedExposure** uses instances of **PmTeFaint3x3** for faint 3x3 event processing, **PmTeFaintBias3x3** for faint-with-bias 3x3 event processing, and **PmTeGraded** for graded event telemetry production (not shown). A detailed description of the **ProcessMode** classes is provided in Section 37.0.

SmContClocking - This is a subclass of **ScienceMode**, and is responsible for configuring and performing a Continuous Clocking bias or data processing run. This class is described in detail in Section 37.0. When configuring the **ProcessMode** pointers, this class uses subclasses of **ProcessMode** to handle different types of Continuous Clocking data processing modes. For raw telemetry mode, it uses instances of the **PmCcRaw** class (not shown). When processing events, **SmContClocking** uses instances of **PmCcFaint1x3** for faint 1x3 event processing, and **PmCcGraded** for graded event telemetry production (not shown). A detailed description of the **ProcessMode** classes is provided in Section 37.0.

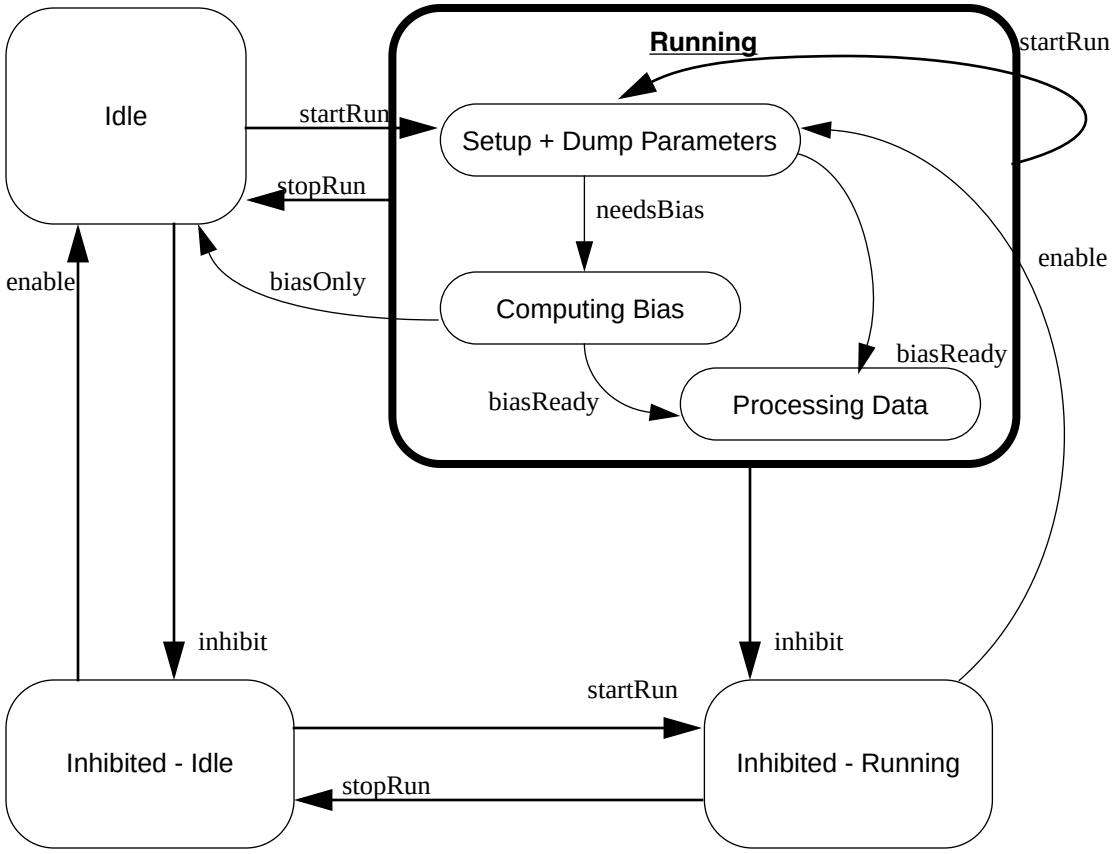
ProcessMode - This abstract class is responsible for processing science data records, produced from a single Front End Processor. This class provides a top level function to process Front End Processor data records (processRecord). This class also provides a collection of member functions (not shown) used to configure the instance for a particular

run. These include functions to indicate and retrieve which CCD is being processed (`setCcdId`, `getCcdId`), set and get the row and column scale factors and other image information (`setGeometry`, `getGeometry`, `setSplitThreshold`), set and get the **`ScienceMode`** from which to obtain telemetry packets (`setMode`, `getMode`), set and get the run's bias and data command and parameter block identifiers (`setRunIdInfo`, `getRunIdInfo`), and set and get the start times of the bias and data runs (`setTimeBias`, `setTimeData`, `getTimeBias`, `getTimeData`). This class also provides a set of common routines used by it and its subclasses. A detailed description of the **`ProcessMode`** classes is provided in Section 37.0.

33.4 Science Manager Behavior

Figure 158 illustrates the overall state behavior of the science manager class.

FIGURE 158. Science Manager state behavior



Initially, the science manager is *idle*. When it receives a `startRun` command, it initiates a new science run. If at any point in the process, the science manager receives a `stopRun` command, the run is terminated, and the manager goes *idle*. The `startRun` indicates whether or not this run should only compute bias, or should process data.

In general, `startRun` requests cause the science manager to start a science run, and `stopRun` requests cause the science manager to terminate the current run. A run may also be terminated once a “compute bias” completes, and no data processing was requested.

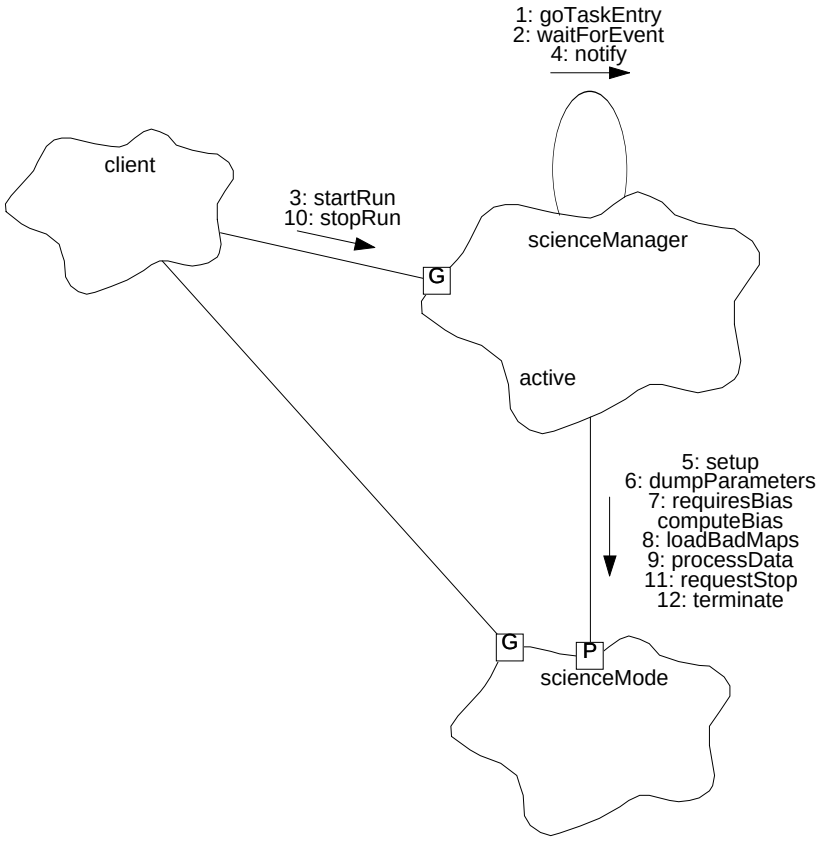
Whenever the science manager is `inhibited`, it maintains the desired running state while it remains *inhibited*, and once *enabled*, it either goes *idle*, or starts a run, depending on the last received `startRun` or `stopRun` command.

33.5 Scenarios

33.5.1 Use 1: Perform a science or bias run

Figure 159 illustrates the overall steps used to conduct a bias and science run.

FIGURE 159. Bias and Science Run



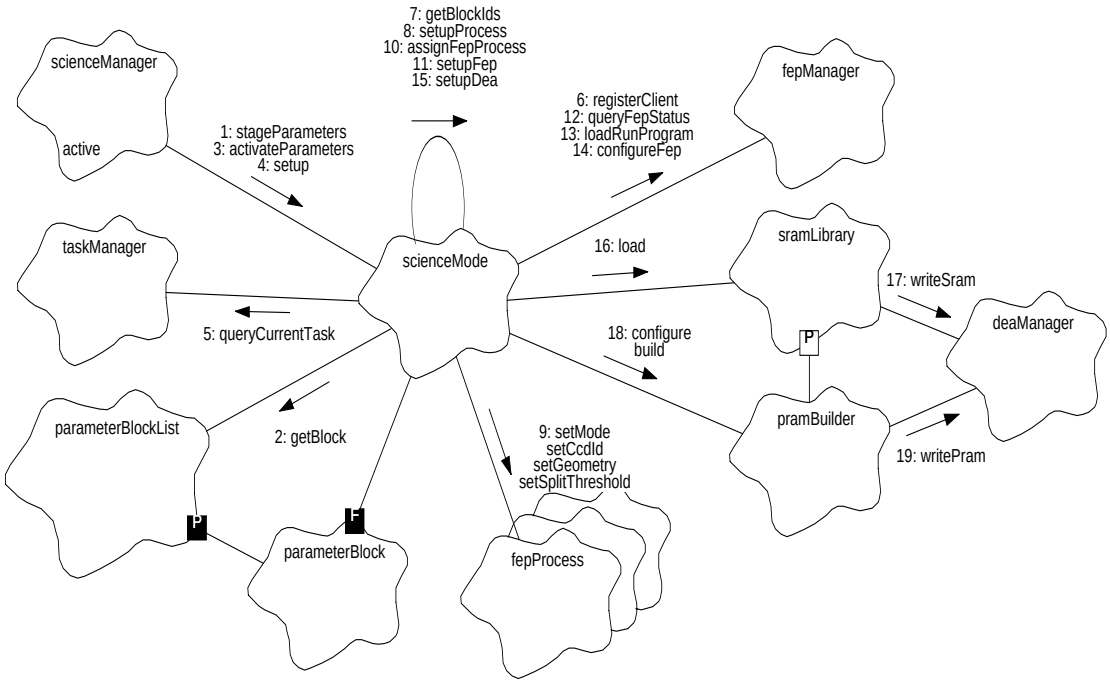
1. Soon after startup, the executive invokes the main function of the Science Manager task, *scienceManager*.goTaskEntry(), which enters an infinite processing loop.
2. goTaskEntry() waits for notification of a command by calling waitForEvent(). Whenever the *taskMonitor* queries the *scienceManager*, *scienceManager* wakes up from *waitForEvent*() and responds to the query using *taskMonitor*.respond() (not shown). This is continued until a command to perform a run is received.
3. A *client* decides to start a science run. It calls *scienceManager*.startRun(), passing the command identifier initiating the run, the parameter block's slot identifier to use, the mode to run, *scienceMode*, and whether or not the run should only compute bias. For the purposes of this scenario, assume that both bias and data processing are to be performed.

4. *scienceManager*.startRun() saves the passed information (such as the parameter block slot id, and whether or not this is a bias-only run), overwriting any pending requests (this occurrence is reported to software housekeeping, but is not shown nor described in this scenario), instructs the mode to copy the requested parameter blocks into its staging area using *scienceMode*.stageParameters() (not shown), and wakes up its task, using notify(). startRun then returns to its caller. Later, waitForEvent() returns, indicating that a request is present.
5. goTaskEntry() copies the request information, tells the mode to activate the staged parameter blocks, using *scienceMode*.activateParameters() (not shown) and calls *scienceMode*.setup() to setup the hardware and software for the run.
6. goTaskEntry() dumps the parameter blocks to telemetry, using *scienceMode*.dumpParameters() (see Section 33.5.3).
7. goTaskEntry() tests if the parameters require a bias computation, using *scienceMode*.requiresBias(). If a bias is required by the parameter block, or if the startRun requested a bias-only run, goTaskEntry() calls *scienceMode*.computeBias() to compute the bias maps on all of the selected Front End Processors (see Section 33.5.4).
8. goTaskEntry() loads the bad pixel or column maps into the Front End Processor's bias maps, using *scienceMode*.loadBadMaps(). NOTE: After adding pixels or columns from the bad pixel or column maps, or deleting the contents of a map, the ground should command a new bias calibration to be performed prior to processing any new science data.
9. If startRun() did not request a bias-only computation, goTaskEntry() calls *scienceMode*.processData() to start data acquisition and processing (see Section 33.5.5). processData() does not return to goTaskEntry() until it is stopped by a call to *scienceMode*.requestStop().
10. Later, while data acquisition and processing are being performed, the *client* tells the *scienceManager* to stop the run, using *scienceManager*.stopRun().
11. stopRun() records the request. If a mode is active, stopRun() calls *scienceMode*.requestStop() to cause the mode to finish up its current data set and stop data processing.
12. Once the data processing stops, *scienceMode*.processData() returns to goTaskEntry(). goTaskEntry() calls *scienceMode*.terminate() to cleanup from the processing and send a final run report to telemetry. goTaskEntry() then repeats its infinite loop, starting from step 2.

33.5.2 Use 2: Setup for a science or bias run

Figure 160 illustrates the high-level steps used to setup for a bias or science run.

FIGURE 160. Science/Bias Run Setup



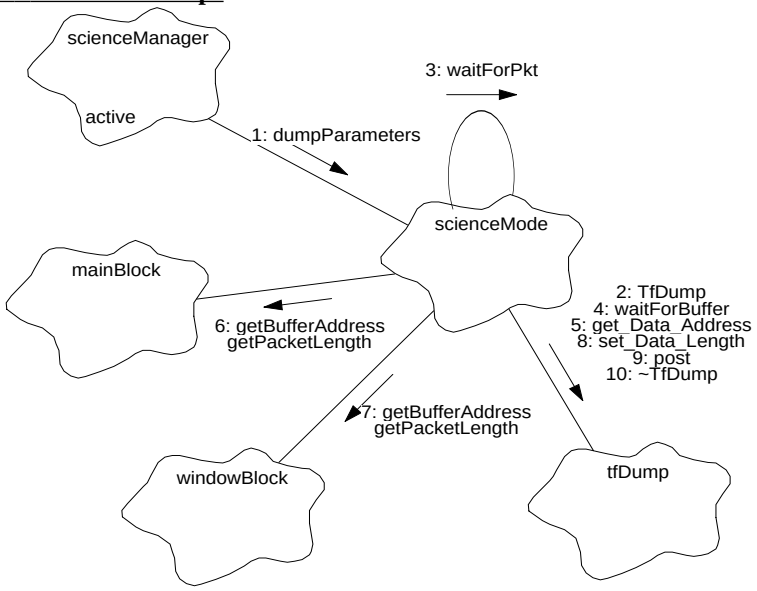
1. When the *scienceManager*'s binding function is instructed to start a new run, it tells the requested mode to stage its parameter blocks, using *scienceMode*.stageParameters().
2. stageParameters() retrieves the requested parameter blocks from the parameter block list and saves them for use later, once the requested mode is ready to run, using *parameterBlockList*.getBlock().
3. Later, once the manager's task is ready, it copies the request into its current state variables, and tells the mode to activate its staged parameter blocks, using *scienceMode*.activateParameters().
4. The *scienceManager* then tells the current mode to prepare for processing, using *scienceMode*.setup().
5. *scienceMode*.setup() asks the *taskManager* which task it is running under, using *taskManager*.queryCurrentTask().
6. setup() tells the *fepManager* which task to notify, and with which events, using *fepManager*.registerClient().
7. setup() then retrieves and stores the identifiers contained within the parameter blocks using getBlockIds().
8. setup() calls setupProcess() to setup the data processing instances for the mode.

9. `setupProcess`() configures each **`ProcessMode`** instance. It establishes itself as the source of telemetry packets, using *`fepProcess`*.`setMode`(). It associates the process with a CCD using *`fepProcess`*.`setCcdId`(), informs the process of the row and column scale factors, the row offset, and the output node configuration, using *`fepProcess`*.`setGeometry`(), and sets the split threshold levels, using *`fepProcess`*.`setSplitThreshold`().
10. `setupProcess`() uses `assignFepProcess`() to install the configured **`ProcessMode`**'s into the process pointer array.
11. `setup`() calls `setupFep`() to load, run and configure each of the Front End Processors.
12. `setupFep`() calls *`fepManager`*.`queryFepStatus`() to determine if the FEP is up and running, and to determine if a new bias computation is needed.
13. If the FEP has power, but is not running, or if a customized Front End Processor code load is required, `setupFep`() calls *`fepManager`*.`loadRunProgram`() to install and run the default or custom FEP program.
14. `setupFep`() uses *`fepManager`*.`configureFep`() to load the mode's FEP parameters into the running Front End Processors.
15. `setup`() uses `setupDea`() to generate and load the Detector Electronics Assembly CCD Controller's sequencer images.
16. `setupDea`() uses *`sramLibrary`*.`load`() to load the standard Sequencer RAM (SRAM) into one or more of the CCD controller sequencer memories.
17. *`sramLibrary`*.`load`() uses *`deaManager`*.`writeSram`() to write to this RAM.
18. `setupDea`() then uses the PRAM builder to construct and load the Program RAM into one or more of the CCD Controller's sequencer memories. The builder is configured using *`pramBuilder`*.`configure`(), and the PRAM program is built and loaded using *`pramBuilder`*.`build`().
19. As the PRAM builder constructs the sequencer's program, it loads the program into sequencer memory using *`deaManager`*.`writePram`().

33.5.3 Use 3: Dump of parameters

Figure 161 illustrates the steps involved in sending the parameter blocks to telemetry.

FIGURE 161. Parameter Dumps

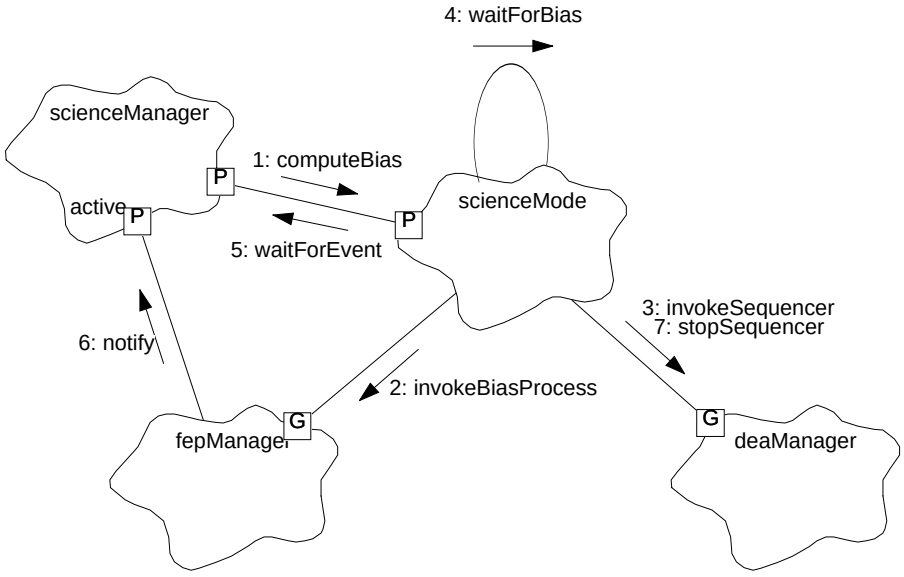


1. The *scienceManager* instructs the setup mode to dump its parameter blocks, using *scienceMode*.dumpParameters().
2. dumpParameters() declares a telemetry form instance, *tfDump* (TfDump()).
3. dumpParameters() passes *tfDump* to waitForPkt() to wait for and associate a telemetry packet with the form.
4. waitForPkt() uses *tfDump*.waitForBuffer() to wait for and allocate a telemetry packet buffer.
5. dumpParameters() uses *tfDump*.get_Data_Address() to obtain the destination pointer for the parameter block within the telemetry packet buffer.
6. It then uses *mainBlock*.getBufferAddress() to get the start of the parameter block buffer, and *mainBlock*.getPacketLength() to get the length of the block. It then copies the parameter block contents directly into the telemetry packet buffer.
7. If a window list is used for the run, dumpParameters() gets the address and length of the list buffer using *windowBlock*.getBufferAddress() and *windowBlock*.getPacketLength(), and appends the window list to the main parameter block within the telemetry packet buffer.
8. dumpParameters() then sets the data length in the telemetry packet using *tfDump*.set_Data_Length().
9. dumpParameters() posts the packet buffer to telemetry, using *tfDump*.post().
10. Once the packet buffer has been posed, dumpParameters() returns, causing the destructor for the form to be called (~TfDump).

33.5.4 Use 4: Perform bias computation

Figure 162 illustrates how a bias computation is initiated.

FIGURE 162. Bias Computations

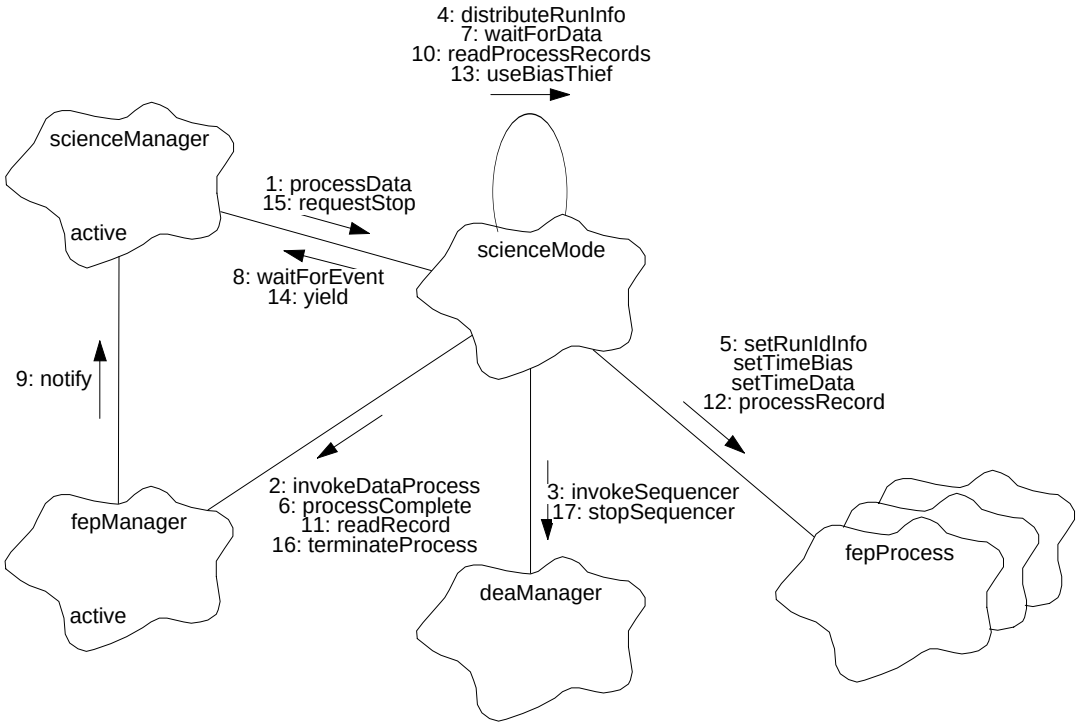


1. *scienceManager* instructs the prepared mode (i.e. *scienceMode*.`setup`() must have already been called) to compute the CCD pixel bias values, using *scienceMode*.`computeBias`().
2. `computeBias`() starts the bias computation process on each of the configured Front End Processors using *fepManager*.`invokeBiasProcess`().
3. `computeBias`() starts the Detector Electronics Assembly CCD Controller Sequencers using *deaManager*.`invokeSequencer`(), and saves the microsecond science time-stamp, latched by the hardware when the command to start the sequencer was issued.
4. `computeBias`() then waits for the bias computation to complete, using `waitForBias`().
5. `waitForBias`() uses the current task to wait to be notified of the completion of the bias computation, using *scienceManager*.`waitForEvent`().
6. Later, once all of the running Front End Processors have completed their respective bias computations, the *fepManager* notifies the installed client, using *scienceManager*.`notify`(), causing `waitForEvent`() to return.
7. `computeBias`() stops the DEA's sequencers, using *deaManager*.`stopSequencer`(). It then calls `loadBadMaps`() (not shown) to load the bad pixel and columns into the bias map memory. `computeBias`() then calls `useBiasThief`() (not shown) to determine if the bias maps are to be telemetered. If so, `computeBias`() calls *biasThief*.`biasReady`() (not shown), causing the *biasThief* to send the maps to telemetry (see Section 33.5.1 for a description of the loading of the bad pixel and column maps).

33.5.5 Use 5: Acquire and process CCD data

Figure 163 illustrates the overall steps used to acquire and process CCD data

FIGURE 163. Data Acquisition and Processing



1. *scienceManager* instructs the *scienceMode* to acquire and process data, using *scienceMode*.processData().
2. processData() starts the Front End Processor's data processing routines, using *fepManager*.invokeDataProcess().
3. processData() starts clocking all of the selected CCDs using *deaManager*.invokeSequencer(), which returns with the start time of the run. NOTE: All of the CCD sequencers start at the same time.
4. processData() distributes the run and start time information to each **ProcessMode** using distributeRunInfo().
5. distributeRunInfo() informs each **ProcessMode** of the command and parameter block identifiers for the most recent bias computation and for the current run using *fepProcess*.setRunIdInfo(). It informs each mode of the most recent bias computation start time using *fepProcess*.setTimeBias(), and of the start time of the current run using *fepProcess*.setTimeData().
6. processData() enters its main processing loop, which terminates on an abort request, error, or when the Front End Processor indicates that processing is complete, via *fepManager*.processComplete().

7. processData() waits for data to arrive from one or more of the Front End Processors, using waitForData().
8. waitForData() uses the current task to wait for the event, *scienceManager*.waitForEvent().
9. Later, one or more of the Front Ends reports that data is available. *fepManager* informs the installed client using *scienceManager*.notify().
10. Once notified, waitForEvent() returns, and processData() calls readProcessRecords() to read and process one or more data records from one of the Front End Processors.
11. readProcessRecords() gets one record from one of the Front Ends using *fepManager*.readRecord().
12. If a record is returned by readRecord(), readProcessRecords() indexes the appropriate data processor, using the Front End identifier, and tells it to process the records, via *fepProcess*[*fepId*].processRecord(). In order to allow the *biasThief* task to run, readProcessRecords() repeats this for no more than 100 records, or until no more records are available (NOTE: The 100 figure can easily be patched).
13. Once readProcessRecords() returns, processData() determines if the bias thief task is being used to trickle bias information to telemetry by calling useBiasThief().
14. If the biases are being sent, processData() tells the current task to yield control to the bias processing task, using *scienceManager*.yield(). This happens for each set of FEP records processed by readProcessRecords(). After yielding, processData() then iterates the data processing loop (see Step 6).
15. At some later time, the *scienceManager* can be commanded to stop the current run. It calls *scienceMode*.requestStop() to attempt to terminate the run. requestStop() then sets an internal flag indicating that a stop has been requested, and notifies the task that a stop has been requested, via *scienceManager*.notify() (not shown).
16. Upon each iteration (Step 6 through Step 15) of the processing loop (starting with Step 6), processData() tests the state of the stop flag. If it is asserted, it tells the Front Ends to finish up their processing, using *fepManager*.terminateProcess(). processData() then continues its processing loop until *fepManager*.processComplete() indicates that all of the Front End's have finished.
17. processData() then stops the CCD clocking, using *deaManager*.stopSequencer(), and returns.

33.6 Class ScienceManager

Documentation:

This class is responsible for managing science operations within the instrument, including storing parameter blocks, initiating science runs, and terminating science runs.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Public Uses:

`ScienceMode`

Public Interface:

Operations:

```
ScienceManager()
goTaskEntry()
inhibit()
isIdle()
startRun()
stopRun()
```

Private Interface:

Has-A Relationships:

ScienceMode* *currentMode*: This references the current science mode being run.

Boolean *currentBiasOnly*: This variable indicates whether or not the current mode should only compute bias. If *BoolTrue*, only a bias computation is being performed. If *BoolFalse*, a science data run (with or without bias) is being run.

unsigned *currentCmdId*: This is the command id which initiated the current run.

ScienceMode* *requestMode*: This references the mode to be run.

Boolean *requestBiasOnly*: This variable indicates whether the requested run should only compute bias.

unsigned *requestCmdId*: This is the command id which is requesting the run.

unsigned *requestBlockId*: This is the slot id of the parameter block to use for the requested run.

DesiredState *desiredState*: This enumeration indicates what state the science manager should be in. It can have the following values:

SCIENCE_IDLE
SCIENCE_RUNNING
SCIENCE_INHIBITED_IDLE
SCIENCE_INHIBITED_RUN

Concurrency: Active

Persistence: Persistent

33.6.1 ScienceManager()

Public member of: **`ScienceManager`**

Arguments:

`unsigned` *`taskid`*

Documentation:

This is the constructor for the ScienceManager. *`taskid`* is the Nucleus RTX Task identifier for the science manager task. This function's initialization list first calls its parent's constructor, `Task`(), passing it *`taskid`*. It then initializes each of its instance variables.

Concurrency: Sequential

33.6.2 goTaskEntry()

Public member of: **`ScienceManager`**

Return Class: **`void`**

Documentation:

This is the main loop of the Science Manager Task. This function is responsible for waiting for state change requests, and responding to these requests.

Semantics:

At the top of the FOREVER loop, if the *`desiredState`* is SCIENCE_RUNNING, use `requestEvent`() to acquire any pending state change or task query events, otherwise, use `waitForEvent`() to block until a such an event occurs. If a task query event is caught, respond using *`taskMonitor`*.`respond`(). If the *`desiredState`* is SCIENCE_RUNNING, copy the *`requestMode`*, *`requestCmdId`* and *`requestBiasOnly`* into the current variables, and then tell the desired mode to activate its staged parameters, using *`mode`*->`activateParameters`(). Then use *`mode`*->`setup`() to setup for a run, and *`mode`*->`dumpParameters`() to dump the parameter blocks for the run. If a bias is needed, use *`mode`*->`computeBias`() to compute the bias map. If data processing is needed, use *`mode`*->`processData`() to perform the data processing phase of the run. Once complete, use *`mode`*->`terminate`() to end the run and zero the current mode.

Concurrency: Synchronous

33.6.3 inhibit()

Public member of: **`ScienceManager`**

Return Class: **`void`**

Arguments:

`Boolean` *on*

Documentation:

This function controls whether or not science runs are inhibited (see Section 33.4). If *on* is *BoolTrue*, science runs are inhibited. If a run is in progress when the function is invoked, the current science run is terminated. New run requests are deferred until runs are enabled. If *on* is *BoolFalse*, then science runs are enabled. If a science run was terminated by an earlier call to `inhibit`(), or was deferred, the run is started.

Semantics:

If *on* is *BoolTrue*: If *desiredState* is IDLE, set state to INHIBITED_IDLE. If *desiredState* is RUNNING, set state to INHIBITED_RUN. If a run is in progress, call *mode*->`requestStop`() to abort the run.

If *on* is *BoolFalse*: If *desiredState* is INHIBITED_IDLE, set state to IDLE. If *desiredState* is INHIBITED_RUN, set state to RUNNING and notify the task that a run request is pending using `notify`().

Concurrency: Synchronous

33.6.4 isIdle()

Public member of: **`ScienceManager`**

Return Class: **`Boolean`**

Documentation:

This function determines whether or not science processing is being performed. If there is no science processing in progress (*currentMode* is NULL, and *desiredState* is not RUNNING), it returns *BoolTrue*. If a science or bias run is in progress, it returns *BoolFalse*.

Concurrency: Synchronous

33.6.5 startRun()

Public member of: **ScienceManager**

Return Class: **Boolean**

Arguments:

ScienceMode& *mode*
unsigned *blockid*
unsigned *cmdid*
Boolean *biasOnly*

Documentation:

This function starts a science run using the passed *mode* to process the setup, bias, run and shutdown states of the run. If *biasOnly* is *BoolTrue*, the manager will setup and compute the bias for the mode and stop. If *biasOnly* is *BoolFalse*, the manager will setup, compute the bias (if mandated by the parameter block), and proceed onto event processing until commanded to stop. *blockid* identifies which parameter slot to use for the run, and *cmdid* is the id of the command packet which is initiating the run.

Semantics:

Copy *mode*, *blockid*, *cmdid* and *biasOnly* into the request parameters. Use *mode*.stageParameters() to load the parameter blocks into the mode's staging area. If the *desiredState* is inhibited, set it to INHIBITED_RUN to indicate that a run is being deferred, otherwise, set it to RUNNING. If a mode is already in progress, use *currentMode*->requestStop() to abort the active run. Use notify() to cause the task to check for the new run request.

Concurrency: Synchronous

33.6.6 stopRun()

Public member of: **`ScienceManager`**

Return Class: **`Boolean`**

Documentation:

This function causes the Science Manager to stop the current science run. If no run is in progress, this function reports it to software housekeeping and returns *`BoolFalse`*. No further action is taken. If a run is underway, this function initiates termination of the run, and returns *`BoolTrue`*.

Semantics:

If the *`desiredState`* is inhibited, set it to INHIBITED_IDLE, otherwise, set it to IDLE. If a mode is already in progress, use *`currentMode`*->`requestStop`() to end the active run.

Concurrency: Synchronous

33.7 Class ScienceMode

Documentation:

This abstract class represents a science processing mode.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

FepManager
BiasThief
DeaManager
TlmForm

Public Interface:

Operations:

```
ScienceMode()
activateParameters()
checkBlock()
computeBias()
dumpParameters()
loadBadMaps()
processData()
requestStop()
requiresBias()
setup()
stageParameters()
terminate()
waitForPkt()
```

Protected Interface:

Has-A Relationships:

`Task*` *`curTask`*: This variable points to the task under which the science mode is running. This variable is assigned during the call to `setup`().

`FilterGrade` *`filterGrade`*[6]: This is a set of filters which select events based on their respective grade codes. See Section 37.15 for a description of this class.

`FilterPh` *`filterPh`*[6]: This is a set of filters which select events based on their corrected amplitude. See Section 37.16 for a description of this class.

`FilterWindow` *`filterWindow`*[6]: This is a set of filters which select events or raw pixels based on their position within the CCD image. See Section 37.14 for a description of this class.

Operations:

```
assignFepProcess()
distributeRunInfo()
getBlockIds()
getFepRequestType()
readProcessRecords()
setupDea()
setupFep()
setupProcess()
useBiasThief()
waitForBias()
waitForData()
waitForEvent()
```

Private Interface:

Has-A Relationships:.

unsigned *startTimeBias*: This variable contains a copy of the science microsecond timestamp at the time the DEA sequencer was started for a bias computation.

unsigned *startTimeData*: This variable contains a copy of the science microsecond time stamp latched when the DEA sequencers were started for science data processing.

ProcessMode* *fepProcess*[6]: This array of **ProcessMode** pointers is indexed by Front End Processor id. Each pointer points to the **ProcessMode** responsible for the corresponding FEP. If 0, then the corresponding FEP is not in-use. This array is set during the call to `setup`().

Boolean *stopRequest*: This variable indicates whether a stop run has been requested. If *BoolTrue*, then the mode will complete the current exposure set and end data processing.

unsigned *cmdIdBias*: This is the id of the command which caused the most recent bias to be computed. NOTE: When executing a bias calibration, this value is identical to *cmdIdCurrent*.

unsigned *blockIdBias*: This is the id of the parameter block used to perform the most recent bias computation. NOTE: When executing a bias calibration, this value is identical to *blockIdCurrent*.

unsigned *cmdIdCurrent*: This is the id of the command which invoked the current run.

unsigned *blockIdCurrent*: This is the id of the parameter block used to configure the current run.

unsigned *winIdCurrent*: This is the id of the window parameter block used for the current run.

RINGREC *record*: This buffers one FEP to BEP data record.

unsigned *pktPollTimeout*: This is the number of ticks to wait for a telemetry packet buffer between checks for abort commands and responses to the task monitor.

Concurrency: Guarded

Persistence: Persistent

33.7.1 ScienceMode()

Protected member of: **`ScienceMode`**

Documentation:

This is the constructor for the **`ScienceMode`**. The initialization statements first set each of the mode's instance variables, setting *`startTimeData`* to 0xffffffff. The body of the constructor then writes 0 into each pointer in the *`fepProcess`* array.

Concurrency: Guarded

33.7.2 activateParameters

Public member of: **`ScienceMode`**

Return Class: **`void`**

Documentation:

This function copies the staged parameter block and window list parameter block into the mode's active copies. This function must be implemented by all subclasses of this class.

Concurrency: Guarded

33.7.3 assignFepProcess()

Protected member of: **`ScienceMode`**

Return Class: **`void`**

Arguments:

`FepId` *`fepId`*
`ProcessMode*` *`process`*

Documentation:

This function installs a FEP process mode, *`process`*, into the processor slot associated with the FEP, *`fepId`*.

Concurrency: Guarded

33.7.4 checkBlock()

Public member of: **ScienceMode**

Return Class: **Boolean**

Arguments:

unsigned *blockid*

Documentation:

This function acquires and verifies the parameter block identified by *blockid*. If the parameter block id is valid, and the indexed block is intact, this function returns *BoolTrue*. If the parameter block id is invalid, or if the parameter block has been corrupted, this function returns *BoolFalse*. This function must be implemented by each subclass of this class.

Concurrency: Synchronous

33.7.5 computeBias()

Public member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function causes the science mode to compute the CCD bias values for all configured CCDs, and load the bad pixel and column maps into the FEP's memory. Calling this function will start clocking the DEA sequencers, and will cause the FEPs to modify their bias maps. If configured to do so, this function will also cause the **BiasThief** task to start sending bias information as soon as it becomes available.

Preconditions:

setup() must be called prior to calling this function.

Postconditions:

Upon successful initiation of a bias computation, *cmdIdBias*, *blockIdBias*, and *startTimeBias* will contain, respectively, the id of the current command, of the current parameter block, and the microsecond timestamp sampled when the DEA was commanded to start clocking the CCDs for the bias computation.

Concurrency: Guarded

33.7.6 distributeRunInfo()

Protected member of: **ScienceMode**

Return Class: **void**

Documentation:

This function distributes the command and parameter block identifiers, and data processing and bias computation start times to each active *fepProcess*.

Preconditions:

`setup`() must have been called prior to calling this function

Concurrency: Guarded

33.7.7 dumpParameters()

Public member of: **ScienceMode**

Return Class: **void**

Documentation:

This function dumps the mode's parameter blocks to telemetry. Each sub-class must implement this function.

Preconditions:

`setup`() must be called prior to calling this function.

Concurrency: Guarded

33.7.8 getBlockIds()

Protected member of: **ScienceMode**

Return Class: **void**

Arguments:

unsigned& *blockId*
unsigned& *winId*

Documentation:

This function returns the identifiers contained within the active parameter block and window list parameter block. On return, blockId contains the parameter block identifier contained within the main parameter block. If a window list was specified, ***winId***, this function stores the identifier contained within the window list block. If no window list is active, the function stores 0xffffffff in the ***winId***. Each subclass of this class must implement this function.

Preconditions:

`setup`() must have been called prior to calling this function

Concurrency: Guarded

33.7.9 getFepRequestType()

Protected member of: **ScienceMode**

Return Class: **int**

Documentation:

This function returns the request type to use when commanding the FEPs to start a run. Each subclass of this class must supply this function.

Concurrency: Synchronous

33.7.10 loadBadMaps()

Public member of: **`ScienceMode`**

Return Class: **`void`**

Documentation:

This function loads the bad pixel and column maps into the configured Front End Processor's pixel bias memory and adjusts the bias map parity bits associated with each affected memory location. Each subclass is required to implement this function.

NOTE: The address of the bias memory is fixed, however, the address of the pixel parity plane must be obtained from the code running on the FEP via a query to the FEP during the call to `setup`().

Preconditions:

`setup`() must be called prior to calling this function. If `computeBias`() is to be called for this run, it must be called prior to calling this function.

Postconditions:

The FEP's pixel bias map values corresponding to bad pixels or columns will be overwritten with codes flagging them as bad. To remove these flags, `computeBias`() must be re-invoked.

Concurrency: Guarded

33.7.11 processData()

Public member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function causes the science mode to acquire and process event or pixel data from the FEPs. This function returns only if the run is aborted, an error occurs, or `requestStop`() is called. This function returns *`BoolTrue`* if data acquisition and processing was started, and *`BoolFalse`* if an error prevented data processing.

Preconditions:

`setup`() must be called prior to calling this function.

Semantics:

Start data processing on the configured FEPs using *`fepManager`*.`invokeDataProcess`(). Then start the CCD sequencers using *`deaManager`*.`invokeSequencer`(), obtaining the science microsecond time-stamp for the run. Then distribute the start-times to each of the data processing modes using `distributeRunInfo`().

Enter the processing loop. The loop terminates on error, or when *`fepManager`*.`processComplete`() indicates that the FEPs are done producing data. In the loop body, check *`stopRequest`*. If it is set, call *`fepManager`*.`terminateProcess`() (once the FEPs are done, *`fepManger`*.`processComplete`() will return *`BoolTrue`*). Call `waitForData`() to let other tasks run while waiting for one or more of the FEPs to produce data. Read and process records from one or more of the FEPs using `readProcessRecords`(). Finally, check if the bias thief is in use, and yield if so.

Once the loop terminates, call *`deaManager`*.`stopSequencer`() to stop clocking images.

Concurrency: Guarded

33.7.12 readProcessRecords()

Protected member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function reads and processes a series of records from one or more of the active FEPs. The function returns *BoolTrue* is successful, and *BoolFalse* on abort request or fatal error.

Preconditions:

This function is intended to be called only during data processing.

Concurrency: Guarded

33.7.13 requestStop()

Public member of: **ScienceMode**

Return Class: **void**

Documentation:

This function signals the mode to stop data processing. It sets *stopRequest* to *BoolTrue*. If *curTask* is not 0, it notifies it that a stop request has been issued.

Concurrency: Synchronous

33.7.14 requiresBias()

Public member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function determines whether or not the configured mode requires a bias computation prior to data processing using the mode's parameter block. If a fresh bias computation is required, it returns *BoolTrue*. If a fresh bias computation is not indicated by the block, it returns *BoolFalse*. Each subclass must implement this function.

Concurrency: Guarded

33.7.15 setup()

Public member of: **ScienceMode**

Return Class: **Boolean**

Arguments:

unsigned *blockid*
unsigned *cmdid*

Documentation:

This function sets up all hardware and software needed to execute a science run or bias computation. *blockid* is the parameter block slot identifier to use for the run. *cmdid* is the id of the command causing the mode to be run.

Concurrency: Guarded

33.7.16 setupDea()

Protected member of: **`ScienceMode`**

Return Class: **`void`**

Documentation:

This function sets up the Detector Electronics Assembly for the current mode. All child classes must implement this function.

Concurrency: Guarded

33.7.17 setupFep()

Protected member of: **`ScienceMode`**

Return Class: **`void`**

Documentation:

This function loads and configures the Front End Processors for the mode. All subclasses must implement this function.

Concurrency: Guarded

33.7.18 setupProcess()

Protected member of: **`ScienceMode`**

Return Class: **`void`**

Documentation:

This function sets up the **`ProcessMode`** instances, for the current run. Each subclass must implement this function.

Concurrency: Guarded

33.7.19 stageParameters

Public member of: **ScienceMode**

Return Class: **void**

Arguments:

unsigned *blockid*

Documentation:

This function makes a local copy of the parameter block identified by *blockid*, and any referenced parameter blocks, in preparation for being commanded to use the blocks later. Each subclass is required to implement this function.

Concurrency: Guarded

33.7.20 terminate()

Public member of: **ScienceMode**

Return Class: **void**

Documentation:

This function causes the science mode to complete its bias/processing operations, perform a post-run integrity check, and telemeter a summary of the run. Each subclass is required to implement this function.

Concurrency: Guarded

33.7.21 useBiasThief()

Protected member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function determines whether or not to use the Bias Thief task to telemeter the contents of the FEP's Bias Map memory. It returns *BoolTrue* if the task should be configured and notified when the maps are ready to be sent, and *BoolFalse* if the bias thief is not used for the current mode. Each subclass is required to implement this function.

Preconditions:

`setup`() must be called prior to calling this function.

Concurrency: Guarded

33.7.22 waitForBias()

Protected member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function suspends the current task until all of the biases have been computed on all of the Front End Processors, or until a command is issued to the mode. This function also periodically wakes up and responds to queries from the *taskMonitor*. It returns *BoolTrue* if the bias computation completed, and *BoolFalse* on an abort or error.

Preconditions:

A bias computation must have been started on the FEPs prior to calling this function.

Concurrency: Guarded

33.7.23 waitForData()

Protected member of: **ScienceMode**

Return Class: **Boolean**

Documentation:

This function waits for notification that data is available from at least one of the enabled FEPs. This function also responds to queries from the *taskMonitor*. The function returns *BoolTrue* if data is available, and *BoolFalse* on error, or if the run has been aborted.

Preconditions:

Data processing must be active on the FEPs.

Concurrency: Guarded

33.7.24 waitForEvent()

Protected member of: **ScienceMode**

Return Class: **Boolean**

Arguments:

unsigned *event*

Documentation:

This function waits for the event (s) indicated by *event*, while also responding to *taskMonitor* queries, and checking for errors and abort requests. If *event* is received with no errors, the function returns *BoolTrue*. If the run was aborted, or an error occurs, the function returns *BoolFalse*.

Concurrency: Guarded

33.7.25 waitForPkt()

Public member of: **ScienceMode**

Return Class: **Boolean**

Arguments:

TlmForm& *form*

Documentation:

This function instructs the telemetry form to obtain a telemetry packet buffer for the *form*. This function periodically checks for task monitor queries and commanded state changes. If a packet is obtained, the function returns *BoolTrue*. If a command or error has aborted the run, the function returns *BoolFalse*.

Concurrency: Guarded

33.8 Class ProcessMode

Documentation:

This class represents a data processing mode for a single FEP.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

TlmForm
ScienceMode

Public Interface:

Operations:

```
ProcessMode()
getCcdId()
getFepId()
getGeometry()
getMode()
getRunIdInfo()
getTimeBias()
getTimeData()
processRecord()
setCcdId()
setFepId()
setGeometry()
setMode()
setRunIdInfo()
setSplitThreshold()
setTimeBias()
setTimeData()
```

Protected Interface:

Has-A Relationships:

`EventExposure` *`expInfo`*: This contains information about the current exposure and run. Its run information, such as the configured split thresholds and image scale and offset factors, is loaded by the call to `setRunInfo`. The function `processRecord` loads exposure-varying information, such as the overclock delta values, by passing FEP produced exposure start and end records to this object's member functions.

Operations:
```
getExposureInfo()
waitForPkt()
```

Private Interface:

Has-A Relationships:

`CcdId` *`ccdid`*: This is the CCD being processed by this instance.

`FepId` *`fepid`*: This is the FEP being processed by this instance.

`ScienceMode`* *`modePtr`*: This variable points to the science mode using the event processor.

`unsigned` *`runCmdIdData`*: This variable contains the command id used to start the science run.

`unsigned` *`runCmdIdBias`*: This variable contains the command id which initiated the most recent bias computation.

`unsigned` *`runPblockData`*: This is the parameter block identifier used to configure the current run.

`unsigned` *`runPblockWin`*: This is the window parameter block identifier being used for the current run.

`unsigned` *`runPblockBias`*: This is the parameter block identifier used to configure the most recent bias computation.

`unsigned` *`runTimeBias`*: This is the start time (latched BEP microsecond science timestamp) of the most recent bias computation.

`unsigned` *`runTimeData`*: This is the start time (latched BEP microsecond science timestamp) of the current run.

Concurrency: Guarded

33.8.1 ProcessMode()

Public member of: **`ProcessMode`**

Documentation:

This is the constructor for the **`ProcessMode`** class. Its initialization statements zero all of its instance variables.

Concurrency: Sequential

33.8.2 getCcdId()

Public member of: **`ProcessMode`**

Return Class: **`CcdId`**

Documentation:

This function returns the CCD identifier being handled by this instance.

Preconditions:

`setCcdId`() must be called prior to using this function.

Concurrency: Synchronous

33.8.3 getFepId()

Public member of: **`ProcessMode`**

Return Class: **`FepId`**

Documentation:

This function returns the FEP identifier being handled by this instance.

Preconditions:

`setFepId`() must be called prior to using this function.

Concurrency: Synchronous

33.8.4 getGeometry()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned& *rowscale*
unsigned& *colscale*
unsigned& *rowoffset*
QuadMode& *nodeSelect*

Documentation:

This function supplies the row and column scale factors, and row offset factor reported via `setGeometry`. `rowscale` is the number of summed rows per pixel row, `colscale` is the number of summed columns per pixel, and `rowoffset` is the number of skipped CCD rows. `nodeSelect` indicates which output node configuration is in use (Full, Diagnostic, AC, or BD).

Preconditions:

`setGeometry`() must be called prior to using this function.

Concurrency: Synchronous

33.8.5 getMode()

Public member of: **ProcessMode**

Return Class: **ScienceMode***

Documentation:

This function returns a pointer to the science mode invoking this data process.

Preconditions:

`setMode`() must be called prior to using this function.

Concurrency: Synchronous

33.8.6 getRunIdInfo()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned& *dataCmdId*
unsigned& *dataBlockId*
unsigned& *dataWinId*
unsigned& *biasCmdId*
unsigned& *biasBlockId*

Documentation:

This function returns the identification information used for the current data process. Upon returning, *dataCmdId* will be set to the id of the command which started the run, *dataBlockId* and *dataWinId* will be set to the parameter block id and window parameter block id used to configure the run. *biasCmdId* will be set to the id of the command which initiated the most recent bias computation, and *biasBlockId* is the parameter block id used to configure that computation.

Preconditions:

`setRunIdInfo`() must be called prior to calling this function.

Concurrency: Synchronous

33.8.7 getTimeBias()

Public member of: **ProcessMode**

Return Class: **unsigned**

Documentation:

This function returns the start time (the latched BEP microsecond science timestamp) of the most recent bias computation.

Preconditions:

`setTimeBias`() must be called prior to using this function.

Concurrency: Synchronous

33.8.8 getTimeData()

Public member of: **ProcessMode**

Return Class: **unsigned**

Documentation:

This function returns the start time (the latched BEP microsecond science timestamp) of the current science data run.

Semantics:

`setTimeData`() must be called prior to using this function.

Concurrency: Synchronous

33.8.9 processRecord()

Public member of: **ProcessMode**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes one FEP block buffer filled with a science record produced by a Front End Processor. *record* contains a copy of the FEP record block. If a run-terminating error occurs, or the run is aborted, this function returns *BoolFalse*. If the records were successfully parsed, or ignored, it returns *BoolTrue*. Each subclass replaces this member function with their own mode-specific code, and will explicitly call this function for records which they don't handle.

Concurrency: Guarded

33.8.10 setCcdId()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

CcdId *ccdId*

Documentation:

This function associates this instance with a particular CCD, *ccdId*

Concurrency: Guarded

33.8.11 setFepId()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

FepId *fep*

Documentation:

This function associates this instance with a particular FEP, *fep*.

Concurrency: Guarded

33.8.12 setGeometry()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned *rowscale*
unsigned *colscale*
unsigned *rowoffset*
QuadMode *nodeSelect*

Documentation:

This function sets the scaling and offset factors for the image being clocked out of the CCD. *rowscale* is the number of summed CCD rows per produced pixel row. *colscale* is the number of CCD columns per produced pixel. *rowoffset* is CCD row index of the first processed CCD row. *nodeSelect* indicates which output node configuration is in use (Full, Diagnostic, AC, or BD).

Concurrency: Guarded

33.8.13 setMode()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

ScienceMode* *mode*

Documentation:

This function associates the science mode instance, *mode*, with this processing instance.

Concurrency: Guarded

33.8.14 setRunIdInfo()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned *dataCmdId*
unsigned *dataBlockId*
unsigned *dataWinId*
unsigned *biasCmdId*
unsigned *biasBlockId*

Documentation:

This function sets the run identification information for the process. *dataCmdId* is the id of the command which initiated the data process, *dataBlockId* is the parameter block id used to configure the run, and *dataWinId* is the parameter block id of the window list used to filter the data. *biasCmdId* is the id of the command which initiated the most recent bias computation and *biasBlockId* is the id of the parameter block used to configure the most recent bias computation.

Concurrency: Guarded

33.8.15 setTimeBias()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned *biasTime*

Documentation:

This function stores the start time (the latched BEP microsecond science timestamp) of the most recent bias computation, *biasTime*.

Concurrency: Guarded

33.8.16 setTimeData()

Public member of: **ProcessMode**

Return Class: **void**

Arguments:

unsigned *dataTime*

Documentation:

This function stores the start time (the latched BEP microsecond science timestamp) of the current science data run, *dataTime*.

Concurrency: Guarded

33.8.17 waitForPkt()

Protected member of: **ProcessMode**

Return Class: **Boolean**

Arguments:

TlmForm& *form*

Documentation:

This function waits tells the *form* to acquire a telemetry packet buffer, using the associated science mode, *mode*.

Concurrency: Guarded

34.0 Timed Exposure PRAM Builder Classes (36-53234 B)

34.1 Purpose

The purpose of the Timed Exposure PRAM Builder is to generate and load CCD-controller Program RAM instructions which clock the CCDs for the Timed Exposure Science Mode.

34.2 Uses

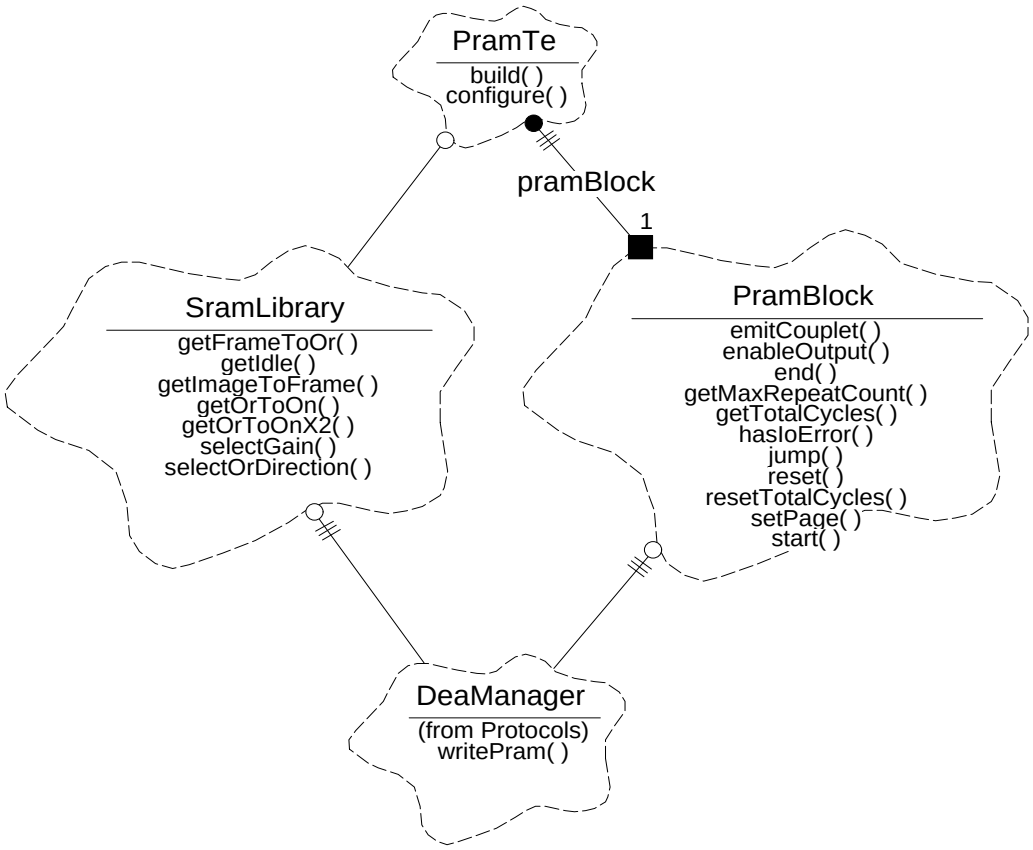
The following lists the uses of the Timed Exposure PRAM Builder:

Use 1:: Build and load Program RAM instructions into a CCD-controller to perform Timed Exposure mode CCD clocking.

34.3 Organization

The following illustrates the relationships used by the Timed Exposure PRAM Builder class, **`PramTe`**, and a utility class which is used by the PRAM builder to manage PRAM Block output, **`PramBlock`**.

FIGURE 164. Timed Exposure PRAM Builder class relationships



`PramTe` - This class is responsible for generating and loading a series of sequencer Program RAM words needed to clock a CCD for Timed Exposure mode. It provides a function which sets the desired clocking parameters (`configure`), and provides a function which generates and loads the clocking sequence into a particular DEA CCD-controller (`build`).

`PramBlock` - This class is responsible for emitting PRAM blocks to the DEA, each containing a header and one or more couplet entries. It is used by PRAM builders, such as **`PramTe`** and **`PramCc`** (see Section 35.0), to load entries into PRAM. It provides functions which reset the instance to the start of PRAM and disable output to the DEA (`reset`), start and end a PRAM block (`start`, `end`), cause jump to a new PRAM page (`jump`), start writing to a new page within PRAM (`setPage`), track the total number of clock cycles used by a block of PRAM (`resetTotalCycles`, `getTotalCycles`), emit a PRAM couplet (`emitCouplet`), obtain the maximum repeat cycles supported by a single couplet (`getMaxRepeatCount`), determine if an error was encountered while writing to the DEA (`hasIoError`), and enable output of PRAM words to the DEA (`enableOutput`).

`DeaManager` - This class is provided by the ***`Protocols`*** class category, and is responsible for managing access to the DEA interface. It provides a function which the builder uses to load words into a CCD-controller's Program RAM (`writePram`).

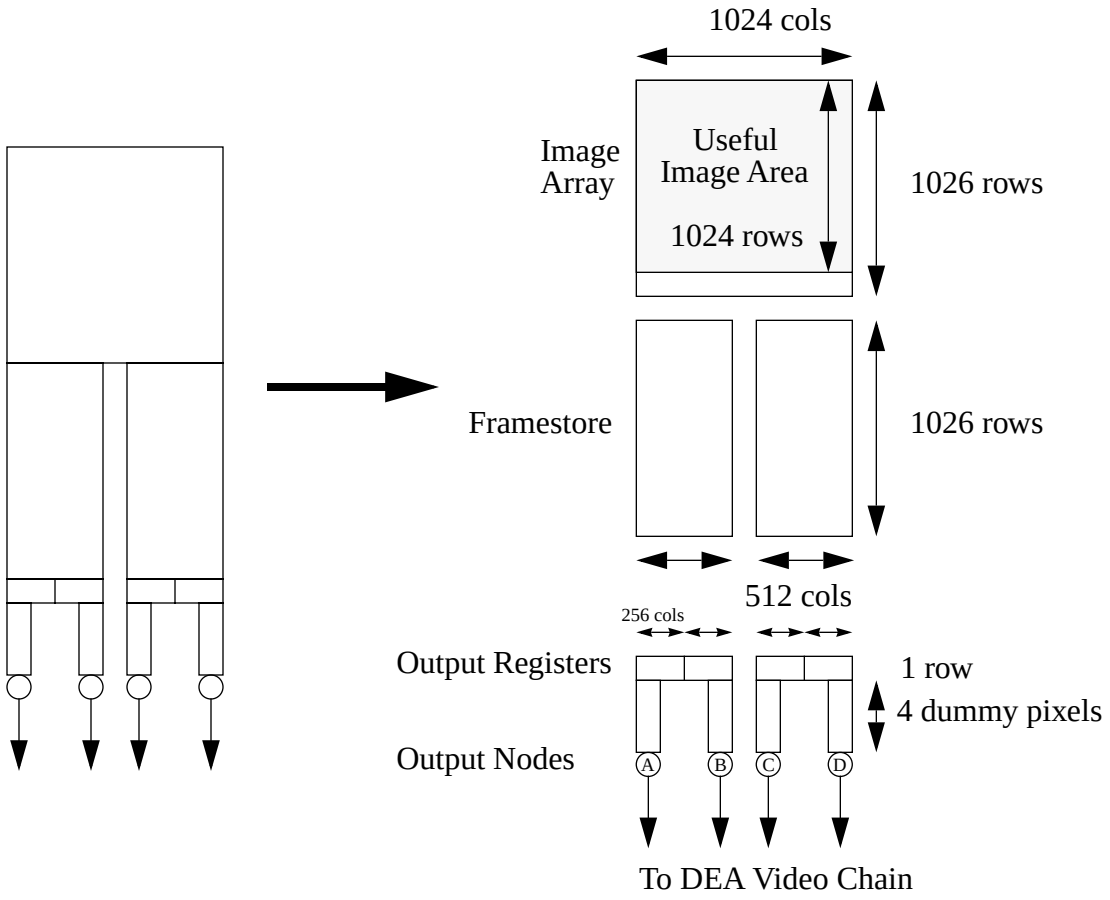
`SramLibrary` - This class is responsible for providing the PRAM builder with the Sequencer RAM (SRAM) locations of various primitives. It provides functions which select certain clocking options to use, such as the attenuated timing for the output register clocking, or the direction to clock the output registers (`selectGain`, `selectOrDirection`). It provides functions which supply the address of an SRAM block which does nothing (`getIdle`), which clock and sample 1 pixel from the output register to the output node (`getOrToOn`), clock and sum two pixels from the output register to the output node (`getOrToOnX2`). It also provides functions which return the starting SRAM block and number of contiguous blocks used to clock one row from the image array to the framestore (`getImageToFrame`), and which clock one row from the framestore into the output registers (`getFrameToOr`).

34.4 PRAM Builder Design Issues

34.4.1 CCD Organization

Figure 165 illustrates a graphical representation of a single CCD. The figure on the left of the illustration presents a simplified picture of the main CCD components, and the figure on the right illustrates the pixel dimensions of each of the components.

FIGURE 165. Graphical CCD Representation

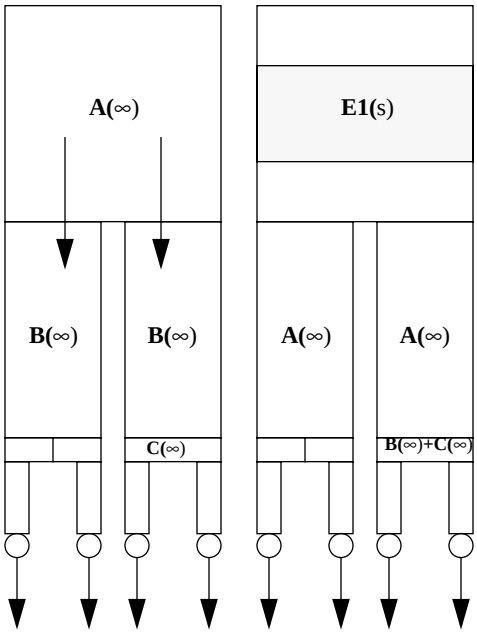


Each CCD consists of an Image Array, a Framestore, two output registers and four output nodes. Only 1024 of the 1026 Image Array rows are used for data acquisition.

34.4.2 Timed Exposure Clocking Sequence

The following illustrates the sequence of operations used to clock CCDs in Timed Exposure Mode (NOTE: For a description of the used SRAM primitives, see Section 34.4.3. The PRAM builder function associated with the action is posted at the bottom of each description).:

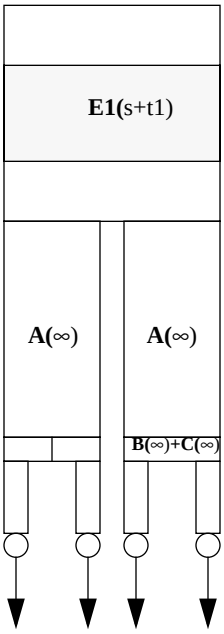
1. Flush image array



Clear charge from the image array by clocking 1026 rows from the image array into the framestore and from the framestore into the output registers. Garbage charge (A) is moved from the image array to the framestore. Meanwhile X-rays, particles, etc. deposit charge into the image array (E1). After the transfer, the “lifetime” of the pixels in the image array (s) is the time it takes to transfer the 1026 rows (i.e. the “smear time”). As the image array is moved into the framestore, whatever charge was in the framestore (B) is added to whatever charge is in the output registers (C).

(see `emitExposure()` and `emitFlushImage()`)

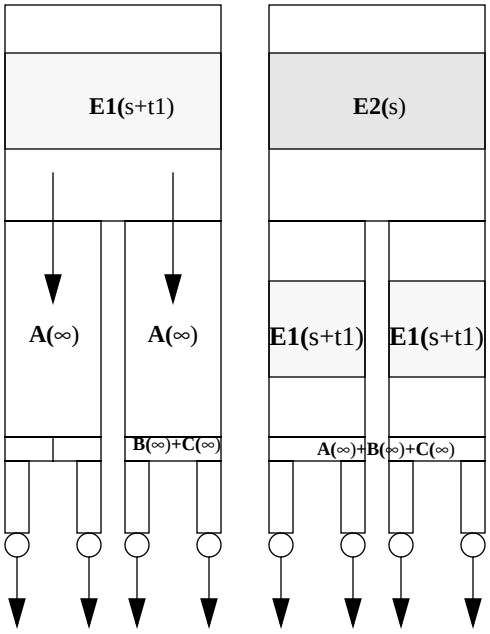
2. Initial exposure



Expose image array for the initial exposure time (t1).

(see `emitIntegrate()`)

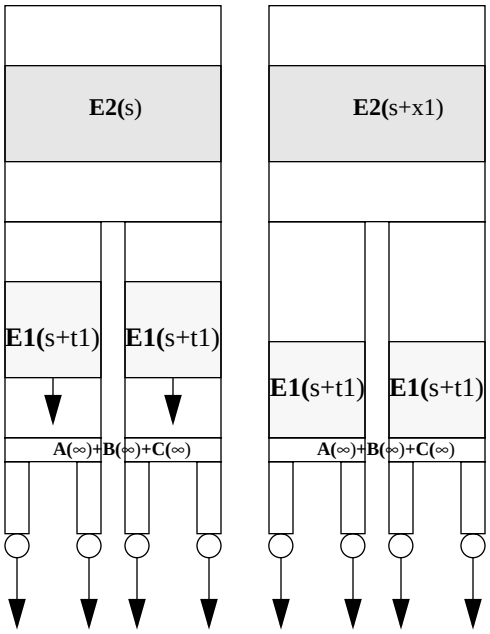
3. Copy image to framestore



Move the charge image (E1) from the image array by clocking 1026 rows from the image array into the framestore and from the framestore into the output registers. Meanwhile X-rays, particles, etc. deposit charge into the next image in the image array (E2). After the transfer the "lifetime" of the first row of new exposure's pixels is the time taken to perform the transfer (s) (i.e. the "smear" time). By convention, this time is not counted against the exposure time of the image.

(see `emitImageToFrame()`)

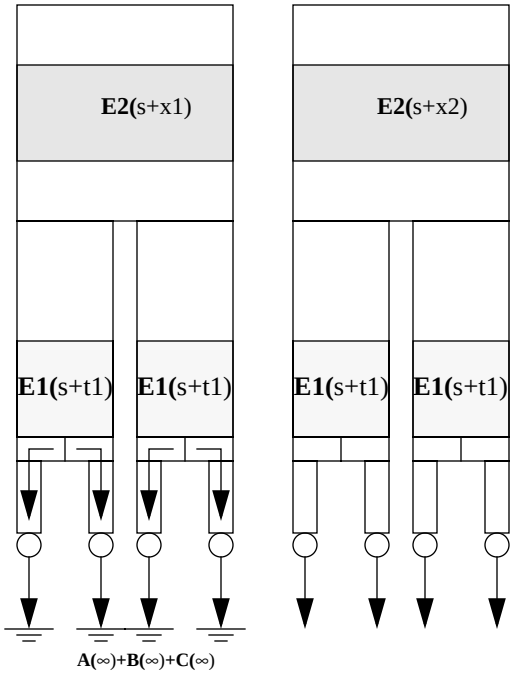
4. Move first row of subarray to bottom of framestore



Clock the framestore until the first row of the desired subarray is adjacent to the output registers. The time to accomplish clocking (x1) is counted as part of the exposure time of the next image (E2).

(see `emitFrameToOr()`)

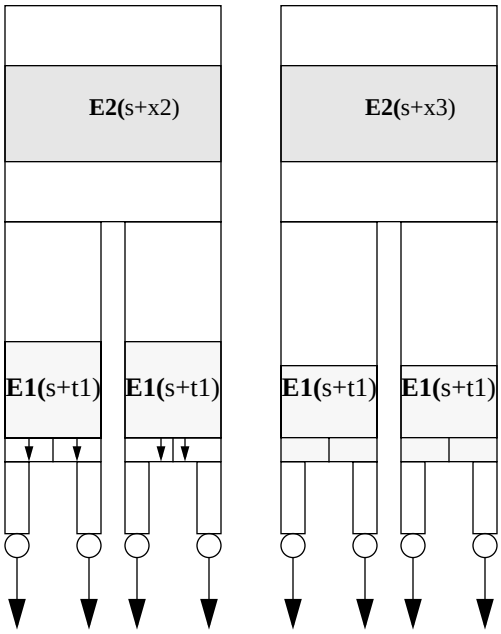
5. Discard accumulated garbage charge in the output register



Clock the output registers and discard the output to clear charge in the registers. The number of times to clock the registers depends on the number of output nodes being used. When in Full or Diagnostic mode, all four output nodes are used, and the registers should be clocked (256 + 4)*2 times. When in AC or BD mode, only two nodes are being used, and (512 + 8)*2 clocks are needed to clear the registers. The time taken to clock the registers is counted against the exposure time of the next image (x2).

(see `emitDiscardOr()`)

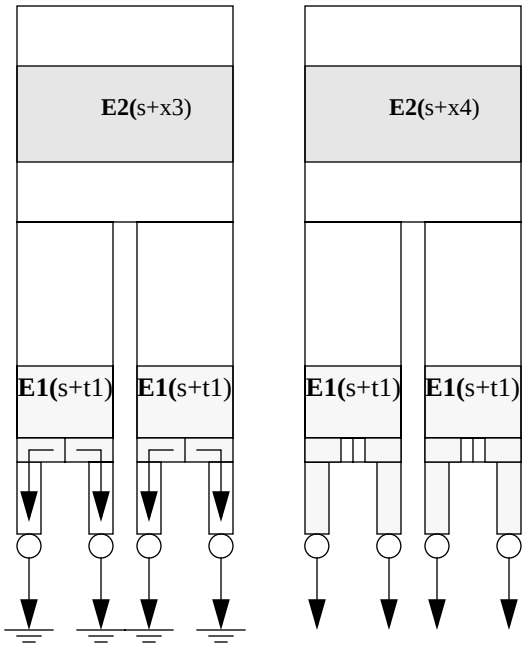
6. Clock first row of image into output register



Clock one row (or two rows if performing 2x2 on-chip summation) from the framestore into the output register. The time taken is counted against the exposure time of the next image(x3).

(see `emitTransferFrame()` and `emitFrameToOr()`)

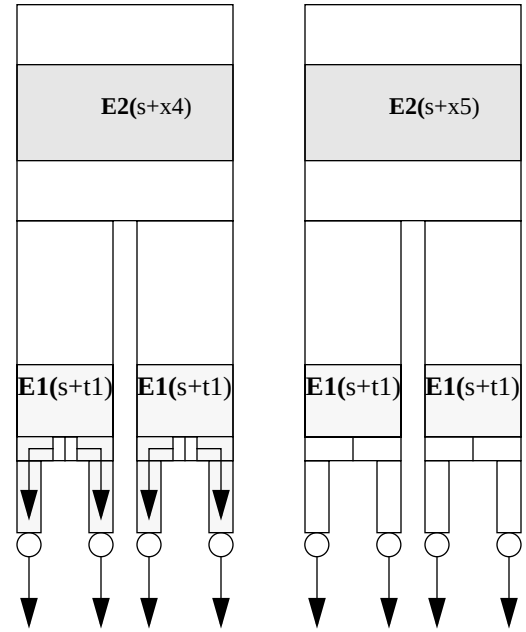
7. Clock and discard first four dummy columns



Clock the four dummy pixels from the output registers to the output nodes and discard the charge. On the last dummy pixel (see Figure 165), emit a pixel code indicating the start of an exposure.

(see `emitTransferFrame()` and `emitDiscardOr()`)

8. Clock row to DEA/DPA for processing

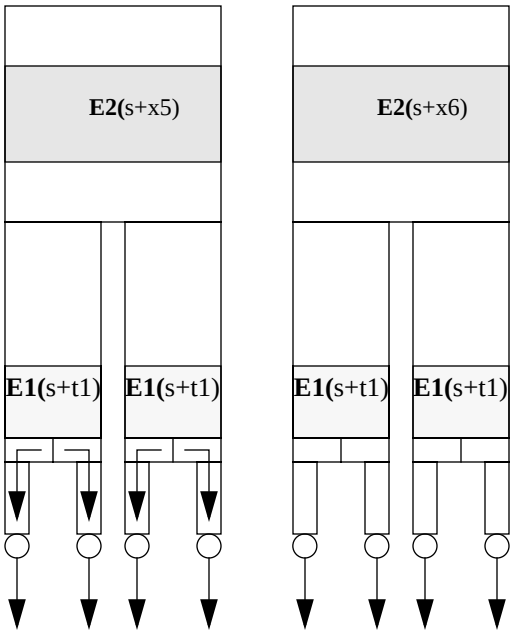


Clock 256 (512 if only using 2 output nodes) pixels from the output registers to the output nodes, sampling each and flagging the pixels to be processed. NOTE: If performing 2x2 on-chip summation, use an SRAM block which clocks and sums two pixels at a time and execute the block 128 times (256 if 2 output nodes are used).

(see `emitTransferFrame()` and `emitOutputPixel()`or, if using 2x2 on-chip summation, `emitSumOr()`)

To DEA Video Chain

9. Overclock output registers

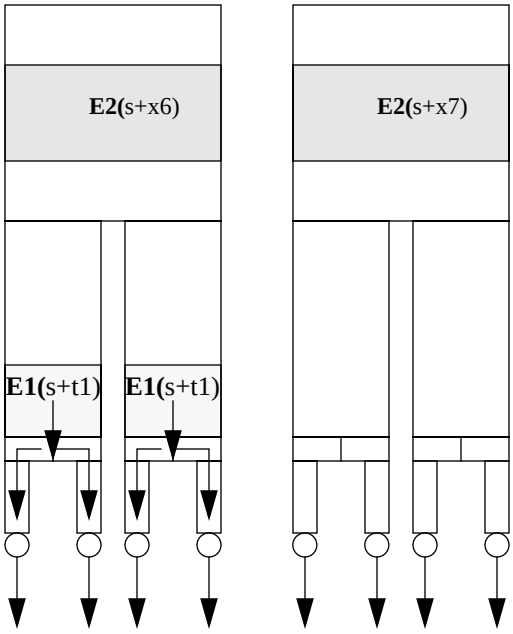


To DEA Video Chain

Clock the output registers a number of times, discarding the output, and then clock the output registers to produce the configured number of overclock pixels.

(see `emitTransferFrame()` and `emitOverclockPixel()`or, if using 2x2 on-chip summation, `emitSumOverclock()`)

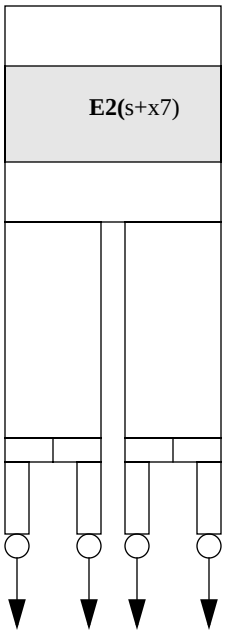
10. Output remaining image



Repeat the steps starting from step 6 until the entire subarray has been output from the framestore. In the last dummy pixel in step 7, however, output a pixel code which indicates the start of a row, rather than the start of an exposure, for all except the first row of the exposure.

(see `emitTransferFrame()`)

11. Process subsequent images



Once an image has been output, the current “lifetime” of the pixels in the next exposure is the smear time (s) plus the total transfer time (x7) of the previous image. Only the transfer time is counted against the exposure time of the next exposure. If the image’s desired exposure time (t2) is greater than the actual time (x7), then process the next image starting from step 2 using (t2 - x7) as the time remaining to expose. If the desired time is identical to the actual, then proceed from step 3. If the desired exposure time is less than the transfer time, discard the accumulated image, and proceed to accumulate a new image by proceeding from step 1.

(see `emitExposure()`)

34.4.3 SRAM Primitives

SRAM consists of a collection of blocks, where each block performs a clocking operation. Some operations can be performed in a single SRAM block, whereas others, due to power constraints, require a series of SRAM blocks. Table 28 lists the SRAM operations used for Timed Exposure Clocking, where each SRAM block takes 1 pixel clock (~10µs) to execute.

TABLE 28. Timed Exposure SRAM Operations

Operation	Number of SRAM Blocks	Description
Integrate	1	Do not perform any clocking operations.[a]
Image to Frame	4	Clock one row from the image array to the framestore, and from the framestore to the output registers.
Frame to OR for Discard	4	Clock one row from the framestore to the output registers, without clocking the image array.
Frame to OR	4	Clock one row from the framestore to the output registers, without clocking the image array.
OR to ON for discard (no summing)	1	Clock one pixel from the output registers to their output nodes and discard the result.[b]
OR to ON discard (x2 summing)	1	Clock two pixels from the output registers to their output nodes and discard the result.
OR to ON sample, standard	1	Clock one pixel from the output registers to their output nodes and sample the result (1 electron/ADU).
OR to ON sample, attenuated	1	Clock one pixel from the output registers to their output nodes and sample the result. Use signal timing to attenuate the signal (4 electrons/ADU).
OR to ON sample, x2, standard	1	Clock two pixels from the output registers to their output nodes, and sum and sample the pixels at the output nodes.
OR to ON sample, x2, attenuated	1	Clock two pixels from the output registers to their output nodes, and sum and sample the pixels at the output nodes.
Reverse OR to ON for discard (no summing)	1	Clock one pixel from the output registers to their opposite output nodes and discard the result.[c]
Reverse OR to ON discard (x2 summing)	1	Clock two pixels from the output registers to their opposite output nodes and discard the result.
Reverse OR to ON sample, standard	1	Clock one pixel from the output registers to their opposite output nodes and sample the result.
Reverse OR to ON sample, attenuated	1	Clock one pixel from the output registers to their opposite output nodes and sample the result. Use signal timing to attenuate the signal.

TABLE 28. Timed Exposure SRAM Operations

Operation	Number of SRAM Blocks	Description
Reverse OR to ON sample, x2, standard	1	Clock two pixels from the output registers to their opposite output nodes, and sum and sample the pixels at the output nodes.
Reverse OR to ON sample, x2, attenuated	1	Clock two pixels from the output registers to their opposite output nodes, and sum and sample the pixels at the output nodes.

a. ‘Integrate’ may be identical to ‘OR to ON with discard’

b. ‘OR to ON with discard’ may be identical to ‘OR to ON sample’. The pixel code is used to tell the DPA to ignore the pixel.

c. The use of only two output nodes, such as in AC and BD mode, is selected via a discrete command to the DEA CCD controller to swap the output register phase clocks.

34.4.4 PRAM Headers and Couplets

This section describes the format of the PRAM words. PRAM consists of collections of word pairs known as ‘couplets.’ Each couplet contains the address of one SRAM block to invoke, a repeat counter indicating the number of times to repeat the block, and a pixel code, which instructs the Front End Processors (FEPs) how to process pixels being clocked while the SRAM block is being invoked. Each invocation of a couplet takes 1 pixel clock period (~10μs).

While PRAM is running, pixel clocks are always being delivered to the FEPs, regardless of whether pixel data is being clocked out of the CCDs or not. The pixel code portion of the couplets indicate how the FEPs interpret the data. Valid pixel data is flagged using a “Valid Pixel” code, and overclock pixels are flagged using an “Overclock” code. All other codes cause the clocked pixel data to be ignored. These include the “Vsync” code, which indicates that a new image has been started, the “Hsync” code, which indicates that a new image row is about arrive, and the “Ignore” code, which indicates that the clocked pixel should be completely ignored by the FEPs.

Each collection of couplets is preceded by a pair of ‘header’ words. The header consists of the number of couplets which follow the block, the number of times to repeat the collection of couplets, and an action to perform once the collection of couplets completes. These actions include: halt the sequencer, continue to the header immediately following the couplet collection, jump to the first PRAM location (restart), and jump to the indicated PRAM page.

The format of this header is as follows:

15	14	13 12	11 0
1	1	Option	PRAM Block Repeat Count
1	0	Page Jump	Couplet Count

PRAM Block Repeat Count .....This specifies the number of times to repeat the entire block minus 1

Option.....This specifies the next sequence option [0: Restart, 1:Continue, 2:Halt, 3:Page Jump]

Couplet Count .....This specifies the number of PRAM word pairs (couplets) following the block minus 1

Page Jump .....If Option is 3, this specifies the PRAM page to jump to

The following illustrates the format of the PRAM couplets within a block:

15	14	13 12	11 5	4	3 0
0	1	SRAM Page Address		0	PixCode
0	0	0	0	Major Cycle Count	

PixCode.....This code is sent to the Front End Processor with each major cycle (pixel), [0:Ignore, 3:Valid Pixel, 4:End of Row (HSYNCH), 8:Start of Image (VSYNCH), 12:Overclock]

SRAM Page Address.....This specifies which block of 64 SRAM blocks should be sequenced during a major cycle.

Major Cycle Count.....This specifies how many times to repeat the selected SRAM block minus 1

PRAM Memory is organized around four contiguous pages, each containing 8192 words. The “jump” option of the PRAM Header causes PRAM to transfer control to the beginning of the specified page once the current PRAM block completes execution.

34.4.5 Clocking Algorithm

The following illustrates the clocking algorithm using pseudo-code, where the **bolded** text indicates operations which utilize the header repeat counter, *italicized* text indicates operations which can be accomplished using a single SRAM block and utilize the couplet repeat counter, and the underlined text indicates looping operations which are “unrolled” by the builder:

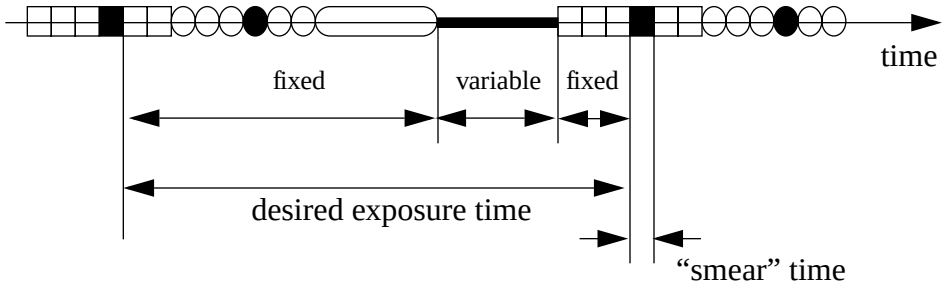
```
PRAM Page 0:
    /* ---- Expose primary exposure time ---- */
    IF primary exposure time is less than transfer time
        Flush image array into framestore
    ENDIF
    Integrate CCD for (primary exposure time - transfer time)

    /* ---- Transfer image to DPA ---- */
    Move 1026 rows from the image array to the framestore
    Locate first row of subarray to first row of framestore
    Discard output register
    REPEAT for each subarray row
        IF 2x2 sum
            Move two rows from framestore to output registers
        ELSE
            Move one row from framestore to output registers
        ENDIF
        Discard 4 dummy output register pixels
        REPEAT for all pixels in output register
            IF 2x2 sum
                Clock, sum and sample 2 pixels from output registers
            ELSE
                Clock and sample 1 pixel from output registers
            ENDIF
        ENDREPEAT
        Discard output register pixels prior to producing overclocks
        REPEAT for each overclock pixel
            IF 2x2 sum
                Clock, sum and sample 2 overclocks from output register
            ELSE
                Clock and sample 1 overclock pixel from output register
            ENDIF
        ENDREPEAT
    ENDREPEAT

    /* ---- Secondary exposures ---- */
    REPEAT duty cycle times
        /* --- Expose Secondary exposure time --- */
        IF secondary exposure time is less than transfer time
            Flush image array into framestore
        ENDIF
        Integrate CCD for (secondary exposure time - transfer time)
        Transfer subarray image to DPA as described above
    ENDREPEAT

    Jump to PRAM Page 0
```

34.4.6 Parallel Transfers with Multiple CCDs

Due to constraints in the ability of the power-supply to deliver enough current to clock large numbers of pixels in all CCDs, each major set of contiguous row transfers from the image array to the framestore, or from the framestore to the output registers must be performed at different times on each CCD. In order to avoid noise from large row transfers on one CCD affecting the Analog-to-Digital conversion on another, all CCDs transfer their respective images to the DEA video system at the same time (NOTE: The time an image spends in the Framestore will be different for each CCD used in a single clocking sequence).

Figure 166 illustrates a one-and-a-half exposure time-line when clocking six CCDs.

FIGURE 166. Single CCD Clocking Sequence



This represents the time during which another CCD is copying its image array to its framestore. The sequence built for this CCD must not clock its image array or framestore during this time.

This represents the time during which the CCD is copying its image array to its framestore. This is the “smear” time of the image and is not counted against the desired exposure time. The sequence built for another CCD must not clock its image array or framestore during this time.

This represents the time during which another CCD is clocking its framestore into its output registers to place the first row of its subarray at the bottom of the framestore. The sequence built for this CCD must not clock its image array or framestore during this time.

This represents the time during which the CCD is clocking the framestore to place the first subarray row into the bottom of the image array. The sequence built for another CCD must not clock its image array or framestore during this time.

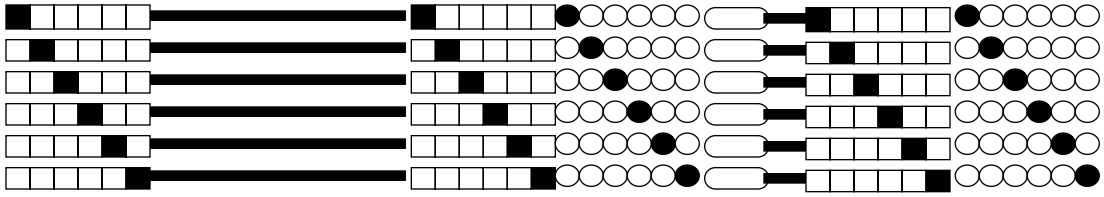
This represents the time during which the CCD is clocking out the subarray image to the DEA video system, and subsequently on to the DPA’s Front End Processors. To minimize the impact of potential cross-talk between subsystems during A/D conversion, all CCDs must do this at the same time.

This represents the additional time that must elapse before the CCD has integrated for the desired exposure time. The PRAM Builder selects this time in order to achieve the observer-specified exposure time.

Figure 167 illustrates various clocking scenarios for CCDs in parallel.

FIGURE 167. Multiple CCD Clocking

Desired Exposure Time >'fixed' clocking/transfer times

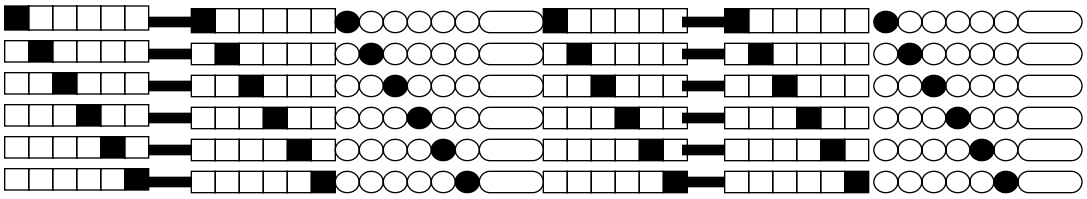


Flush initial image and integrate next image

Transfer image and integrate next image

Transfer image and integrate next image

Desired Exposure Time < 'fixed' clocking/transfer times



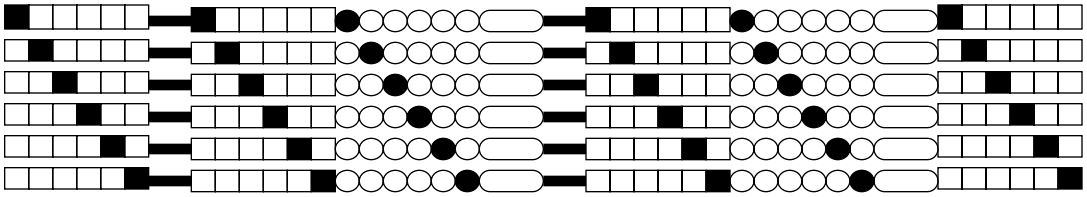
Flush initial image and integrate

Transfer image

Flush and integrate

Transfer image

Short Exposure then Long Exposure then Short Exposure



Flush and integrate short exposure

Transfer short exposure image and integrate long exposure

Transfer long exposure image

Flush an integrate short exposure

In order to schedule each portion of the clocking operation, the PRAM builder must compute the time required to perform the image array to framestore transfer for a CCD, the time to position the first row of a subarray to the bottom of the framestore, and the time to transfer the subarray image to the DEA video system, and subsequently, on to the DPA's FEPs.

The time to transfer one row from the image array to the framestore is determined by the number of SRAM major cycles needed to perform the operation. The time to transfer the entire image array, is computed by multiplying the answer by 1026 rows. The minimum exposure time supported by the instrument is this result, multiplied by one less than the number of CCDs.

The time to position the subarray to the end of the framestore is the number of SRAM major cycles needed to clock 1 row in the framestore times the location of the first row (row 0) of the subarray (relative to CCD image row 0) plus 2 (to handle the two unused rows at the bottom of the image array).

The time taken to transfer the subarray image from the framestore to the DPA is a function of the number of rows in the subarray, the output node configuration, and the on-chip summing selection. In order to reduce the possibility of error when and if changes are made in how images are transferred from the framestore, the builder uses a two-pass approach when building images (see Section 34.5.2).

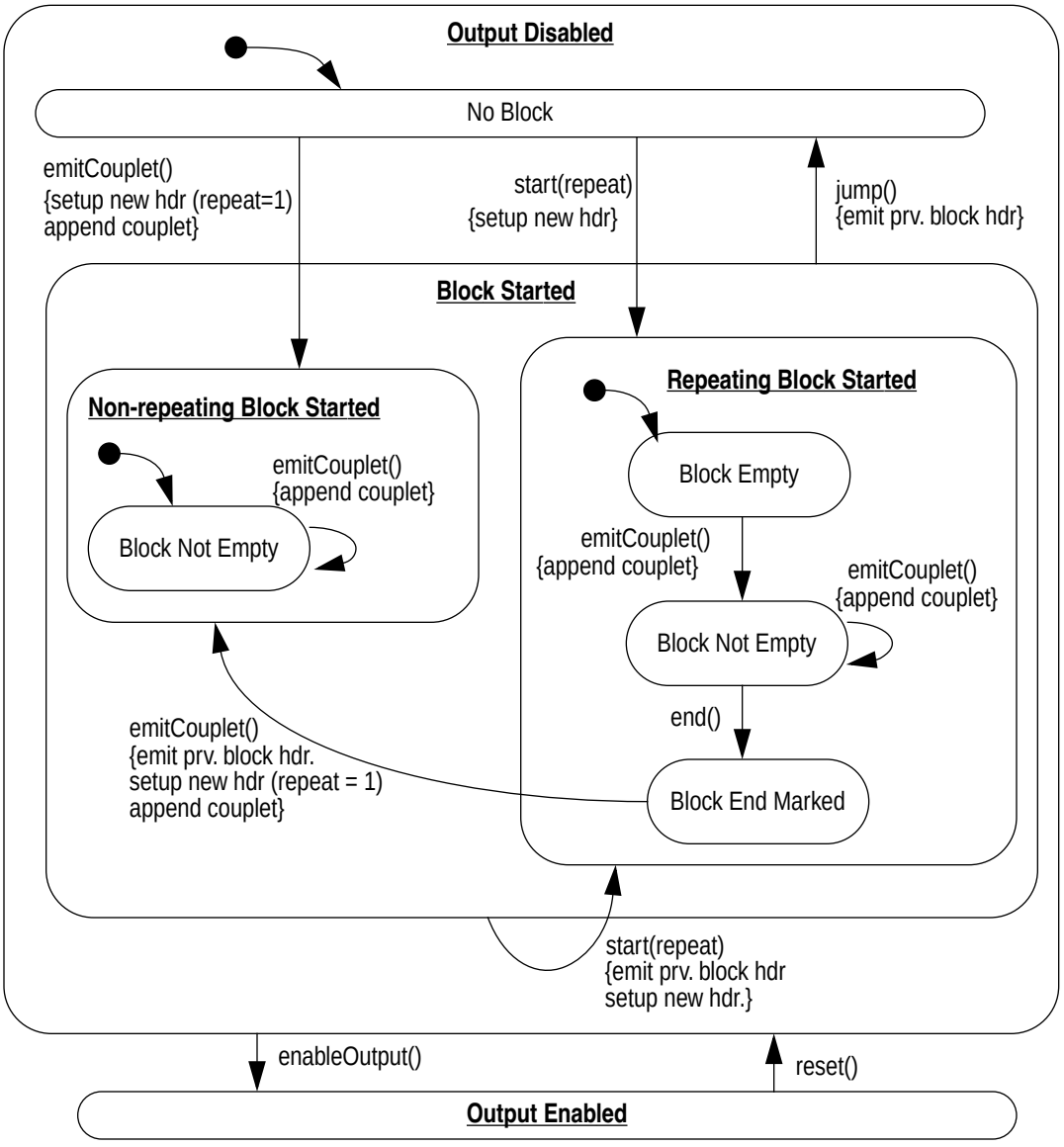
34.5 Scenarios

34.5.1 Using the PramBlock class

This section describes the design and use of the **`PramBlock`** class. This class is responsible for generating PRAM blocks (see Section 34.4.4), and loading these blocks into the DEA.

Figure 168 illustrates the overall behavior of the **`PramBlock`** class, in the form of a finite-state machine. The substates and transitions within the Output Disabled and Output Enabled superstates are identical, except that while in the Output Disabled state, the **`PramBlock`** class does not issue write requests to the **`DeaManager`**.

FIGURE 168. PramBlock Behavior



Clients use the **`PramBlock`** class to construct a series of non-repeating and repeating PRAM blocks. Since all useful PRAM loads within ACIS consist of an infinite loop, and since such loops require a page-jump, the client terminates the with a call to `jump`(), which forms the main infinite-loop of the PRAM sequence, and closes out the last PRAM block in the load.

For example, to build and emit a sequence on PRAM Page 0 which consists of a series of non-repeating blocks, followed by a block which repeats its contents 10 times, followed by a jump to the beginning of the page, the client issues the following call-sequence:

```
pramBlock.reset()
pramBlock.enableOutput()
pramBlock.setPage(0)
pramBlock.emitCouplet(...)
pramBlock.emitCouplet(...)
.
.
.
pramBlock.start(10)
pramBlock.emitCouplet(...)
pramBlock.emitCouplet(...)
.
.
.
pramBlock.end()
pramBlock.jump(0)
```

This produces a PRAM load which consists two PRAM blocks. When run, the first block will execute once and will then continue to the next adjacent block. The second will be executed 10 times, and once complete, will jump to the block at the start of PRAM Page 0 (which contains the first block).

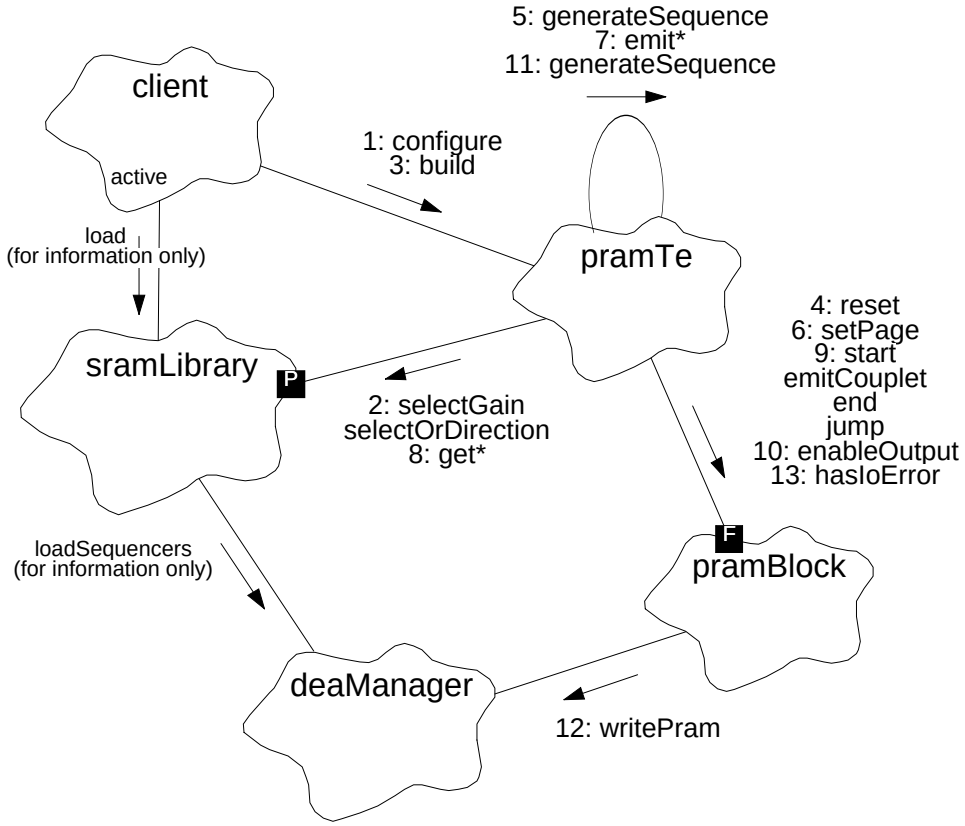
In order to determine the total number of PRAM clock cycles used by a PRAM block, the client adds calls to `resetTotalCycles`() and `getTotalCycles`(). For example, the following determines the time it will take to execute the 2nd PRAM block, in terms of PRAM cycles:

```
pramBlock.reset()
pramBlock.enableOutput()
pramBlock.setPage(0)
pramBlock.emitCouplet(...)
pramBlock.emitCouplet(...)
.
.
pramBlock.resetTotalCycles()
pramBlock.start(10)
pramBlock.emitCouplet(...)
pramBlock.emitCouplet(...)
.
.
.
pramBlock.end()
pramBlock.getTotalCycles()
pramBlock.jump(0)
```

34.5.2 Use 1: Build and load PRAM to perform Timed Exposure CCD clocking

Figure 169 illustrates the use of the **`PramTe`** class to generate a PRAM load for Timed Exposure mode clocking (NOTE: The `load` call to *`sramLibrary`*, and `loadSequencer` call to *`deaManager`* are for reference information only. For more detail, see Section 36.0).

FIGURE 169. Build PRAM Load



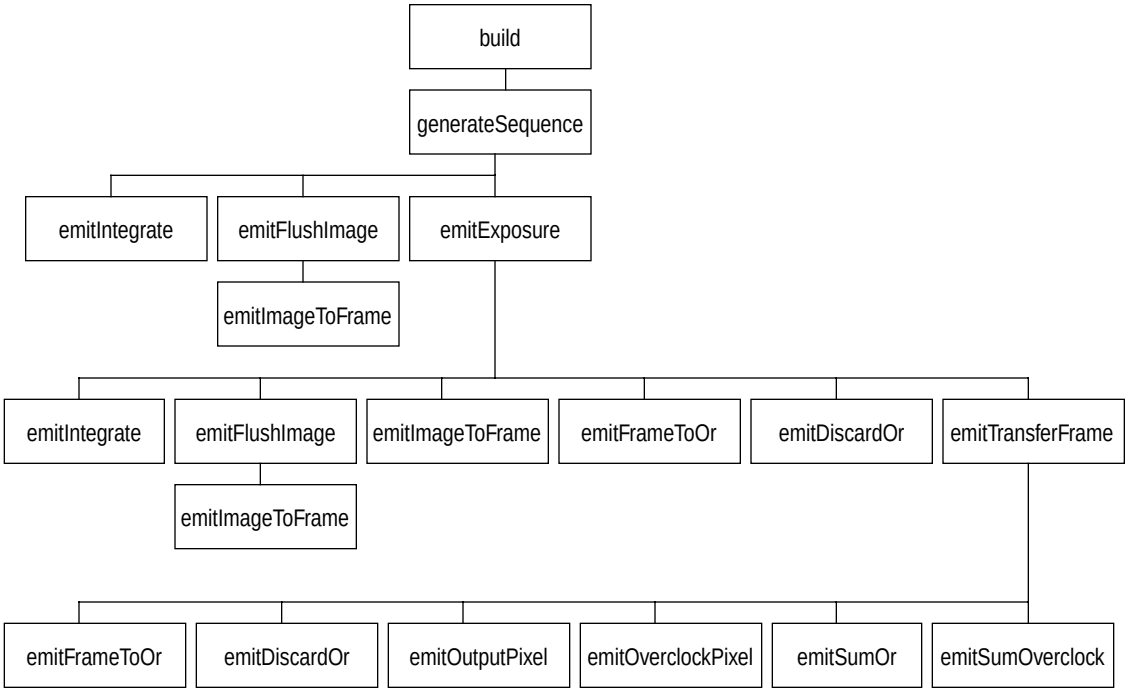
The **`PramTe`** builder uses a two-pass approach to constructing PRAM. The first pass is used to calculate the minimum integration time (in terms of clock cycles) which the current parameter set can support without an added flush of the image-array. The second pass is used to actually produce and store the generated sequence.

1. The *`client`* configures the builder, passing the clocking parameters, and pointer to the *`sramLibrary`* to *`pramTe`*.`configure`().
2. `configure`() initializes the *`pramTe`*'s instance variables and then selects which gain and output register clocking direction primitives to use, using *`sramLibrary`*.`selectGain`() and *`sramLibrary`*.`selectOrDirection`(), respectively.
3. The *`client`* then passes the CCD Id and parallel-transfer phase slot number to *`pramTe`*.`build`() to generate the clocking sequence, and load the sequence into the DEA.

4. `build`() first invokes *`pramBlock`*.`reset`() to set the current page to PRAM Page 0, and to disable output to the DEA.
5. `build`() then computes the minimum non-flushed integration time, given the configured clocking parameters using `generateSequence`().
6. `generateSequence`() uses *`pramBlock`*.`setPage`() to set the initial PRAM page to use.
7. `generateSequence`() uses a variety of *`pramTe`*.`emit*`() functions to emit the various phases of the clocking sequence (see Figure 170 for a call structure chart).
8. The `emit*`() functions use the various *`sramLibrary`*.`get*`() functions to obtain the address and block count of the SRAM primitives used to perform certain functions.
9. The `emit*`() functions use *`pramBlock`*.`start`()/`emitCouplet`()/`end`()/`jump`() functions to generate the PRAM blocks (i.e. header followed by one or more couplets) needed to clock the CCD (see Section 34.5.1for the use of the **`PramBlock`** class). Since DEA output had been disabled by the earlier call to *`pramBlock`*.`reset`(), no words are actually written to the DEA.
10. Once the minimum non-flushed integration time is computed, *`pramTe`*.`build`() calls *`pramBlock`*.`enableOutput`() to enable writes to the DEA.
11. `build`() then proceeds with pass-2 by calling `generateSequence`() to generate and load the final clocking sequence, this time with appropriate image-array flush sequences.
12. During the second pass of `generateSequence`(), the *`pramBlock`* functions use *`deaManager`*.`writePram`() to load the PRAM words into the DEA.
13. If an error is encountered during a write, *`pramBlock`* sets an internal I/O error flag and disables further writes, but allows the sequence generation to complete. *`pramTe`*.`build`() then can examine whether or not an I/O error was encountered during the second-pass using *`pramBlock`*.`hasIoError`(), and reports the condition in its return to the calling client.

Figure 170 illustrates a structure chart, indicating the functional hierarchy within the **`PramTe`** class:

FIGURE 170. PramTe Class Internal Structure Chart



34.6 Class PramTe

Documentation:

This class generates the Program RAM sequence needed to perform Timed Exposure clocking of a CCD.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

SramLibrary

Public Interface:

Operations:
`PramTe()`
`build()`
`configure()`

Protected Interface:

Operations:
`emitDiscardOr()`
`emitExposure()`
`emitFlushImage()`
`emitFrameToOr()`
`emitImageToFrame()`
`emitIntegrate()`
`emitOutputPixel()`
`emitOverclockPixel()`
`emitSumOr()`
`emitSumOverclock()`
`emitTransferFrame()`
`generateSequence()`

Implementation:

Has-A Relationships:

unsigned *`exposureTime`*[2]: Primary and Secondary Exposure Times, in units of 1/10th's of a second. Copied by `configure`().

unsigned *dutyCycle*: Number of *exposureTime*[1]'s per *exposureTime*[0]'s, starting with *exposureTime*[0]. Copied by configure().

unsigned *rowStart*: First row of subarray, not including the two dummy rows at the start of the image (see *dummyRows*). Copied by configure().

unsigned *rowEnd*: Last row of subarray. Copied by configure().

QuadMode *qMode*: Output node configuration, either Full, Diagnostic, AC, or BD. Copied by configure().

unsigned *overclockPairs*: Number of pairs of overclock pixels to output. Copied by configure().

Boolean *sumFlag*: 2x2 On-chip summing flag. If *BoolFalse*, no on-chip summing, if *BoolTrue*, sum pairs of columns and rows. Copied by configure().

Boolean *gain4*: Determines whether or not to use attenuated SRAM timing. If *BoolFalse*, use 1 electron:1 ADU SRAM, if *BoolTrue*, use 4 electrons:1 ADU SRAM timing. Copied by configure().

const unsigned *arrayRows*: Number of non-summed rows in the CCD Image and Frame Store.

const unsigned *nodeCols*: Number of un-summed columns per output node (assuming all 4 nodes in use).

const unsigned *dummyCols*: Number of dummy columns between the output register and the output node.

const unsigned *dummyRows*: Number of rows to skip between each image.

const unsigned *clocksPerExpTime*: Number of pixel clocks per input exposure time units (i.e. number of clocks per 0.1 second).

const unsigned *ocDummy*: This contains the number of dummy pixels to clock out of the output register prior to clocking out overclock pixels.

const unsigned *orDiscardCount*: This contains the number of times the entire Output Register should be clocked to flush charge from the registers.

unsigned *exposureClocks*[2]: These indicate the primary and secondary exposure times, expressed in units of pixel clocks.

unsigned *shiftCols*: This specifies the number of unsummed columns per output node (not including dummy columns).

unsigned *summedRows*: This is the total number of summed rows to clock out of the CCD.

unsigned *summedShiftCols*: This specifies the number of summed columns to shift out for each row.

unsigned *xfrSlots*: This contains the number of chips being configured. This value is used to stagger the parallel transfers of each CCD being used.

unsigned *xfrTime*: Total transfer time for an exposure, including time taken by other CCDs to clock their image array to their framestore, time for all CCDs to position their first subarray row to the bottom of their framestore, and time taken to transfer the subarray image to the video chains. This time does NOT include the time taken by the current CCD to transfer its image array to its framestore (‘smear’ time).

Boolean *configured*: This flag indicates whether or not configure() has been called. If not, this field contains *BoolFalse*. If configure() has been called the field is *BoolTrue*.

unsigned *curslot*: This specifies the current parallel transfer slot to use for the selected CCD-controller board.

SramLibrary* *sramLibrary*: This is a pointer to the **SramLibrary** instance used to use for the build (as passed into configure()).

unsigned *xfrImageToFrame*: This contains the number of clock cycles needed for one CCD to transfer all rows from its image array to its framestore (i.e. # SRAM cycles/row * 1026 rows).

unsigned *xfrSubarrayStart*: This contains the number of clock cycles needed for one CCD to position the first row of its subarray to the bottom of the framestore (i.e. # SRAM cycles/row * (subarray start + 2 dummy rows).

PramBlock *pramBlock*: This is used by the Timed Exposure Builder to emit blocks of PRAM couplets.

Concurrency: Guarded

Persistence: Transient

34.6.1 PramTe()

Public member of: **PramTe**

Documentation:

This function is the constructor for the **PramTe** class. This function initializes the instance variables for the **PramTe** class and sets the *configured* state variable to *BoolFalse*. The body of the constructor invokes *pramBlock*.reset() to place the PRAM Block writer instance into a known state.

Concurrency: Guarded

34.6.2 build()

Public member of: **PramTe**

Return Class: **Boolean**

Arguments:

CcdId *ccdId*
unsigned *phaseslot*

Documentation:

This function builds a PRAM sequence and loads the sequence into the DEA CCD-controller specified by *ccdId*. *phaseSlot* specifies the parallel transfer slot used by the CCD. *phaseSlot* can range from 0 to one minus the number of configured CCDs (see configure()). The function returns *BoolTrue* if the sequence is successfully built and loaded into the DEA. It returns *BoolFalse* if an error was encountered while trying to write to the DEA.

Preconditions:

configure() must have been called (*configured* == *BoolTrue*), and *phaseslot* must be less than the total number of boards specified by the most recent call to configure() (*phaseslot* < *xfrSlots*).

Semantics.

Store *phaseslot* into the private *curslot* variable.

Pass 1: Disable builder output to the DEA and reset *pramBlock* using *pramBlock*.reset(). Compute and store the image transfer time by passing *BoolTrue* to generateSequence(). generateSequence() stores the computed time in *xfrTime*.

Pass 2: Enable DEA output by passing *ccdId* to *pramBlock*.enableOutput(). Generate and emit the clocking sequence for the CCD by passing *BoolFalse* to generateSequence(). Check for output errors using *pramBlock*.hasIoError(). If there is an error, return *BoolFalse*, otherwise, return *BoolTrue*.

Concurrency: Guarded

34.6.3 configure()

Public member of: **PramTe**

Return Class: **Boolean**

Arguments:

unsigned *boardcnt*
unsigned *expPrimary*
unsigned *expSecondary*
unsigned *expDuty*
unsigned *rowstart*
unsigned *rowend*
Boolean *sum2x2*
QuadMode *quadrants*
unsigned *ocPairs*
Boolean *attenuate*
const SramLibrary& *sramlib*

Documentation:

This function configures the Timed Exposure PRAM builder. *boardcnt* is the total number of CCDs being clocked, *expPrimary* is the primary exposure time, *expSecondary* is the secondary exposure time, and *expDuty* is the number of *expSecondary* exposures per *expPrimary* exposures. *rowstart* is the index of the first CCD row of the subarray, and *rowend* is the index of the last row. *sum2x2* indicates whether or not to perform 2x2 on-chip summation. *quadrants* indicates the output node configuration and clocking direction (Full, Diagnostic, AC, or BD). *ocPairs* indicates the number of pairs of overclock pixels to generate. attenuate indicates whether or not to use 4:1 SRAM video timing. *sramlib* is a reference to the SRAM library to use for the run.

Preconditions:

rowstart must be less than or equal to *rowend*, which must be less than 1024. It is the caller's responsibility to configure swapped Output register clocks for AC and BD quadrant clocking modes.

Semantics.

Copy the parameters into the corresponding instance variables, configure the SRAM library's gain and output register direction, and compute derived clocking count information. Finally, set *configured* to *BoolTrue*.

Concurrency: Guarded

34.6.4 emitDiscardOr()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *npix*
PramPixCode *pixcode*

Documentation:

Emit PRAM couplets which clock *npix* columns from the output register into the output node, each time, discharging the output node, effectively discarding the pixels. *pixcode* indicates the pixel code to use when discarding the pixels.

Semantics.

If *npix* is greater than or equal to 2, start by discarding pixels two at a time by fetching the SRAM block using *sramLibrary*->getOrToOnX2(), and emitting the couplet with a repeat count of *npix*/2, using *pramBlock*.emitCouplet(). Pass *pixcode* as the pixel code to emit while performing the clocking. Then adjust *npix* to contain the remainder (NOTE: Use of a remainder is a hook in case faster SRAM blocks become available).

If *npix* is not zero (in this case, it will be 1), get the SRAM block to discard 1 pixel using *sramLibrary*->getOrToOn(), and emit the couplet to repeat *npix* times, using *pramBlock*.emitCouplet() and passing *pixcode* as the pixel code to emit while performing the clocking.

Concurrency: Guarded

34.6.5 emitExposure()

Protected member of: **`PramTe`**

Return Class: **`unsigned`**

Arguments:

`unsigned` *`expclocks`*

Documentation:

Emit PRAM instructions to integrate the CCD for a period of time, and then transfer the exposed image to the DPA. The total integration of the image is specified by *`expclocks`*. This function returns the number of clock cycles taken to clock the image from the Framestore to the DPA (i.e. the minimum integration time that can be supported without a flush of the image-array).

Preconditions:

If not performing a timing run, *`xfrTime`* must contain the number of clock cycles used to transfer an image from the framestore to the DPA. This function assumes that previous cycles have integrated the CCD for this period of time.

Semantics.

If *`xfrTime`* is greater than *`expclocks`*, emit instructions to align the CCD to its designated parallel transfer slot, using `emitIntegrate`(), and then flush the contents of the image array, using `emitFlushImage`(). If not, subtract *`xfrTime`* from *`expclocks`*, and add in the additional time required to align the CCD to its parallel transfer slot. If the result is not zero, emit instructions to integrate the CCDs for the remaining time using `emitIntegrate`().

After the CCD has been integrated and aligned to the appropriate parallel transfer slot, emit instructions to transfer the image array into the framestore using `emitImageToFrame`(). Start measuring the number of cycles taken to transfer the image to the DPA, using *`pramBlock`*`.resetTotalCycles`(). Use `emitIntegrate`() to align the CCD with its sub-array positioning parallel transfer slot. Once aligned, call `emitFrameToOr`() to emit instructions to clock the framestore until the first row of the subarray is in the first row of the framestore, call `emitIntegrate`() to align the CCD with the other CCDs for A/D conversion, and then call `emitDiscardOr`() to discard the contents of the output registers.

Then emit instructions to transfer the image from the framestore to the DPA

using `emitTransferFrame`().

Compute the minimum number of integration cycles that can be supported without an intervening flush. First, obtain the number of cycles since the image-to-frame transfer using *`pramBlock`*.`getTotalCycles`(). Then compute the additional number of cycles needed to align the next image-to-frame transfer to the CCD's parallel transfer slot, and add it to the measured value. Return the result to the caller.

Concurrency: Guarded

34.6.6 emitFlushImage()

Protected member of: **`PramTe`**

Return Class: **`void`**

Arguments:

`unsigned` *`nrows`*

Documentation:

Emit PRAM instructions to transfer *`nrows`* rows from the image array to framestore and from the framestore to the output register for discard. This function calls `emitImageToFrame`() to emit the instructions for the transfer.

Concurrency: Guarded

34.6.7 emitFrameToOr()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *nrows*

Documentation:

Emit PRAM couplets which transfer *nrows* from the framestore into the output register. The charge from the rows will be summed together and with whatever charge exists in the output register.

Semantics.

Call *sramLibrary*->getFrameToOr() to obtain the starting index of the series of blocks which transfer one row from the framestore to the output register, and to obtain the number of adjacent blocks need for the transfer. If only 1 block is required, call *pramBlock*.emitCouplet(), passing the index of the block, and a repeat count of *nrows* to emit the PRAM instructions for the transfer.

If more than one block is needed and more than one row is being output, call *pramBlock*.start() to start a repeating PRAM block, which, when executed, repeats *nrows* times. If only one row is being output, a repeating block is not required. Then call *pramBlock*.emitCouplet() for each SRAM block in the series. If a repeating PRAM block was started, call *pramBlock*.end() to finish of the block (NOTE: The header of the block won't actually be written until the next *pramBlock*.start()/ emitCouplet()/jump() operation is invoked).

Concurrency: Guarded

34.6.8 emitImageToFrame()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *nrows*

Documentation:

Emits PRAM directives to transfer *nrows* from the image array to the framestore and from the framestore to the output register.The charge from the framestore rows will be summed together with whatever charge exists in the output register.

Semantics.

Call *sramLibrary*->getImageToFrame() to obtain the starting index of the series of blocks which transfer one row from the image array to the framestore and from the framestore to the output register, and to obtain the number of adjacent blocks need for the transfer. If only 1 block is required, call *pramBlock*.emitCouplet(), passing the index of the block, and a repeat count of *nrows* to emit the PRAM instructions for the transfer.

If more than one block is needed and more than one row is being output, call *pramBlock*.start() to start a repeating PRAM block, which, when executed, repeats *nrows* times. If only one row is being output, a repeating block is not required. Then call *pramBlock*.emitCouplet() for each SRAM block in the series. If a repeating PRAM block was started, call *pramBlock*.end() to finish of the block (NOTE: The header of the block won't actually be written until the next *pramBlock*.start()/ emitCouplet()/jump() operation is invoked).

Concurrency: Guarded

34.6.9 emitIntegrate()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *expcounts*

Documentation:

This function emits an SRAM block which lets the CCD sit still (expose) for *expcounts* pixel clock cycles. (NOTE: If *expcounts* is 0, then no action is taken and the function immediately returns).

Preconditions:

expcounts must be less than PRAM_MAX_HDR_REPEAT times PRAM_MAX_BLK_REPEAT (assuming that both constants have a value of 4096, and that 1 SRAM block executes in 10µs, the maximum interaction time supported by one call to this function is about 170 seconds).

Semantics.

This function calls *sramLibrary*->getIdle() to obtain the index of the SRAM block to use. It then compares *expcounts* to the maximum cycles handled by a single couplet, PRAM_MAX_BLK_REPEAT. If *expcounts* is greater than PRAM_MAX_BLK_REPEAT, start a repeating block using *pramBlock*.start(), passing *expcounts* divided by PRAM_MAX_BLK_REPEAT as the number of times to repeat the couplets. Then call *pramBlock*.emitCouplet(), passing PRAM_MAX_BLK_REPEAT as the number of times to execute the block, and call *pramBlock*.end() to mark the end of the repeating block.

If *expcounts* was less than PRAM_MAX_BLK_REPEAT, or the remainder was not zero, call *pramBlock*.emitCouplet() to continue to integrate for the required time.

Concurrency: Guarded

34.6.10 emitOutputPixel()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *npix*

Documentation:

Emit PRAM couplets which transfer *npix* pixels from the output register to the output node and sample each pixel at the output node. This function calls *sramLibrary*->getOrToOn() to obtain the index of the SRAM block to use to output and sample a single pixel, and then calls *pramBlock*.emitCouplet(), passing *npix* as the repeat count, and using the PIXCODE_PIXEL pixel code to indicate to the DPA that the pixel is part of the image data.

Concurrency: Guarded

34.6.11 emitOverclockPixel()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *npix*

Documentation:

Emit PRAM instructions to clock and sample *npix* overclock pixels from the output register. This function calls *sramLibrary*->getOrToOn() to obtain the index of the SRAM block to use to output and sample a single pixel, and then calls *pramBlock*.emitCouplet(), passing *npix* as the repeat count, and using the PIXCODE_OVERCLOCK pixel code to indicate to the DPA that the pixel is part of the image's overclock data.

Concurrency: Guarded

34.6.12 emitSumOr()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *npix*

Documentation:

Emit PRAM instructions to clock and sum two pixels from the output register into the output node, and then sample and transmit the result to the DPA. *npix* indicates the number of pairs of pixels to clock. This function calls *sramLibrary*->getOrToOnX2() to obtain the index of the SRAM block to use to sum and sample two pixels at a time, and then calls *pramBlock*.emitCouplet(), passing *npix* as the repeat count, and using the PIXCODE_PIXEL pixel code to indicate to the DPA that the pixel is part of the image data.

Concurrency: Guarded

34.6.13 emitSumOverclock()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *npix*

Documentation:

Emit PRAM instructions to clock and sum pairs of overclock pixels from the output register. *npix* indicates the number of pairs of overclock pixels to clock. This function calls *sramLibrary*->getOrToOnX2() to obtain the index of the SRAM block to use to sum and sample two pixels, and then calls *pramBlock*.emitCouplet(), passing *npix* as the repeat count, and using the PIXCODE_OVERCLOCK pixel code to indicate to the DPA that the pixel is part of the image's overclock data.

Concurrency: Guarded

34.6.14 emitTransferFrame()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

unsigned *nrows*

Documentation:

Emit PRAM instructions to transfer *nrows* of an image from the framestore to the DEA/DPA via the output registers and output node.

Semantics.

No 2x2 Summation:

Transfer first row of image from framestore to output register, using `emitFrameToOr`(), discard first 4 dummy pixels by calling `emitDiscardOr`(). On the fourth dummy pixel, indicate the start of a new image, by passing a PIXCODE_VSYNCH pixel code. Output the contents of the first row by calling `emitOutputPixel`(). If any overclocks where configured, discard the first few overclocks using `emitDiscardOr`(), and then output the desired number of overclocks using `emitOverclockPixel`(). Indicate the end of the row by passing PIXCODE_HSYNCH to `emitDiscardOr`(). If there is more than 1 row remaining in the image, start a repeat block header using *`pramBlock`*`.start`(). If there is only one row remaining, a repeating block is not needed. Transfer the subsequent rows from the framestore to the output register in the same fashion as the first row, except that no PIXCODE_VSYNCH is required. If a repeating block was created to handle the subsequent rows, call *`pramBlock`*`.end`() to end it.

2x2 Summation:

Perform most of the same steps described above, however, clock two rows at a time from the framestore to the output register, and use `emitSumOr`(), rather than `emitOutputPixel`(), to output pixels, and use `emitSumOverclock`(), rather than `emitOverclockPixel`(), to output overclocks.

Concurrency: Guarded

34.6.15 generateSequence()

Protected member of: **PramTe**

Return Class: **void**

Arguments:

Boolean *timingRun*

Documentation:

This function runs through the PRAM generation process. If *timingRun* is *BoolTrue*, then the function computes and stores the minimum un-flushed integration time, into xfrTime, otherwise, xfrTime is not modified. Upon returning, *xfrTime* contains the transfer time for an exposure.

Semantics.

Generate the main exposure sequence in PRAM Page 0. Call pramBlock.setPage() to start writing to PRAM Page 0. Generate the PRAM load for the primary exposure, by passing *exposureClocks*[0] to *emitExposure*(). If measuring timing (*timingRun* is *BoolTrue*), store the result in *xfrTime*. If the secondary exposure time is different than the primary time, and the *dutyCycle* parameter is not zero, produce a series of secondary exposures by calling emitExposure() *dutyCycle* times, passing *exposureClocks*[1] as the desired exposure time. NOTE: The transfer time of the primary and secondary exposures must be identical.

Once the main exposures have been produced, form the main loop of the sequence by producing a jump back to PRAM Page 0 using *pramBlock*.jump().

Concurrency: Guarded

34.7 Class PramBlock

Documentation:

This class represents a header/couplet array within PRAM. It provides functions which incrementally build and load a block into a DEA CCD Controller.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

DeaManager

Public Interface:

Operations:
```
PramBlock()
emitCouplet()
enableOutput()
end()
getMaxRepeatCount()
getTotalCycles()
hasIoError()
jump()
reset()
resetTotalCycles()
setPage()
start()
```

Protected Interface:

Operations:
```
buildCouplet()
buildHeader()
emitHeader()
```

Private Interface:

Has-A Relationships:

unsigned *_headerAddr*: This is the address in PRAM of the block header.

enum BlockState *_state:* This represents the state of the current PRAM block. BLK_NONE indicates that no block has been started, BLK_STARTED indicates that a PRAM block has been started, and, BLK_ENDED indicates that the end of a PRAM has been marked.

unsigned *_coupletCnt*: This represents the number of couplets emitted within the block so far.

unsigned *_coupletCycles*: This represents the number of cycles taken to execute the block's cycles once.

unsigned *_repeat*: This is the number of times the block is to be repeated.

unsigned *_totalCycles*: This represents the total number of effective execution cycles since the last call to `resetTotalCycles`().

unsigned *_curAddr*: This is the next address to write to within PRAM.

CcdId *_ccd*: This specifies which CCD controller is being loaded by the builder.

Boolean *_enabledOutput*: This indicates if output to the DEA is enabled. If *BoolFalse*, don't write to the DEA. If *BoolTrue*, then generated headers and couplets are written to the DEA.

Boolean *_ioError*: This indicates if a write to the DEA failed. If *BoolFalse*, then no errors have been encountered since the last `reset`(). If *BoolTrue*, then a write error occurred, and further writes have been disabled until the next `reset`().

Concurrency: Guarded

Persistence: Transient

34.7.1 PramBlock()

Public member of: **PramBlock**

Documentation:

This function is the constructor for the **PramBlock** class. This function calls `reset`() to initialize the instance variables and disable output to the DEA.

Concurrency: Guarded

34.7.2 buildCouplet()

Protected member of: **PramBlock**

Return Class: **void**

Arguments:

unsigned *sramaddr*
PramPixCode *pixcode*
unsigned *repeat*
unsigned short[2] *couplet*

Documentation:

This function formats a PRAM couplet word pair, storing the formatted words into *couplet*. *sramaddr* is the SRAM block address to invoke, *pixcode* is the pixel code to emit during execution of the block and *repeat* is the number of times to repeat the block.

Concurrency: Guarded

34.7.3 buildHeader()

Protected member of: **PramBlock**

Return Class: **void**

Arguments:

PramOp *operation*
PramPage *page*
unsigned *couplets*
unsigned *repeat*
unsigned short[2] *header*

Documentation:

This function formats a PRAM block header and stores the formatted words in *header*. *operation* is the action to take once the PRAM couplets have been processed and *page* is the PRAM page to jump to if operation specifies a jump. *couplets* is the total number of couplets in the block, and *repeat* is the number of times to repeat the block.

Concurrency: Guarded

34.7.4 emitCouplet()

Public member of: **PramBlock**

Return Class: **void**

Arguments:

unsigned *sramaddr*
PramPixCode *pixcode*
unsigned *repeat*

Documentation:

This function builds a couplet word pair and writes the pair into the DEA's PRAM. *sramaddr* is the SRAM address of the SRAM block to invoke for the couplet. *pixcode* is the pixel code to associate with the pixels produced while the block is being invoked. *repeat* is the number of times to repeat the block.

Concurrency: Guarded

34.7.5 emitHeader()

Protected member of: **PramBlock**

Return Class: **void**

Arguments:

PramOp *operation*
PramPage *page*

Documentation:

This function writes the block header to PRAM. *operation* is the action to take once the block has been executed, and *page* specifies which page to jump to if the operation warrants it.

Concurrency: Guarded

34.7.6 enableOutput()

Public member of: **PramBlock**

Return Class: **void**

Arguments:

CcdId *ccd*

Documentation:

This function enables output to the CCD indicated by *ccd*. To disable output, call `reset`().

Concurrency: Guarded

34.7.7 end()

Public member of: **PramBlock**

Return Class: **void**

Documentation:

This function closes the current block, but does not write the header. The header is written upon the next call to start or jump.

Concurrency: Guarded

34.7.8 getMaxRepeatCount()

Public member of: **PramBlock**

Return Class: **unsigned**

Documentation:

This function returns the maximum number of repeat cycles that a single couplet can support.

Concurrency: Guarded

34.7.9 getTotalCycles()

Public member of: **PramBlock**

Return Class: **unsigned**

Documentation:

This function returns the total number of effective execution cycles since the last call to `resetTotalCycles`().

Concurrency: Guarded

34.7.10 hasIoError()

Public member of: **PramBlock**

Return Class: **Boolean**

Documentation:

This function indicates whether or not an I/O error was encountered when loading PRAM. *BoolFalse* indicates that no error has been encountered since the last call to reset(). *BoolTrue* indicates that a write failed, and that further writes have been disabled.

Concurrency: Guarded

34.7.11 jump()

Public member of: **PramBlock**

Return Class: **void**

Arguments:

PramPage *page*

Documentation:

This function writes the header to the current block, issuing a jump operation to the PRAM page indicated by *page*.

Concurrency: Guarded

34.7.12 reset()

Public member of: **PramBlock**

Return Class: **void**

Documentation:

This function resets the instance's state variables, disables output to the DEA, clears the I/O error flag, and sets the current PRAM address to the start of PRAM.

Concurrency: Guarded

34.7.13 resetTotalCycles()

Public member of: **PramBlock**

Return Class: **void**

Documentation:

This function zeros the total execution cycle counter.

Concurrency: Guarded

34.7.14 setPage()

Public member of: **PramBlock**

Return Class: **void**

Arguments:

PramPage *page*

Documentation:

This function sets the current PRAM address to the location corresponding to *page*.

Concurrency: Guarded

34.7.15 start()

Public member of: **PramBlock**

Return Class: **void**

Arguments:

unsigned *repeat*

Documentation:

This function flushes any previous block, and opens a new block at the current PRAM address. The opened block is configured to be executed *repeat* times.

Concurrency: Guarded

35.0 Continuous Clocking PRAM Builder Class (36-53235 B)

35.1 Purpose

The purpose of the Continuous Clocking PRAM Builder is to generate and load CCD-controller Program RAM instructions which clock the CCDs for the Continuous Clocking Science Mode.

35.2 Uses

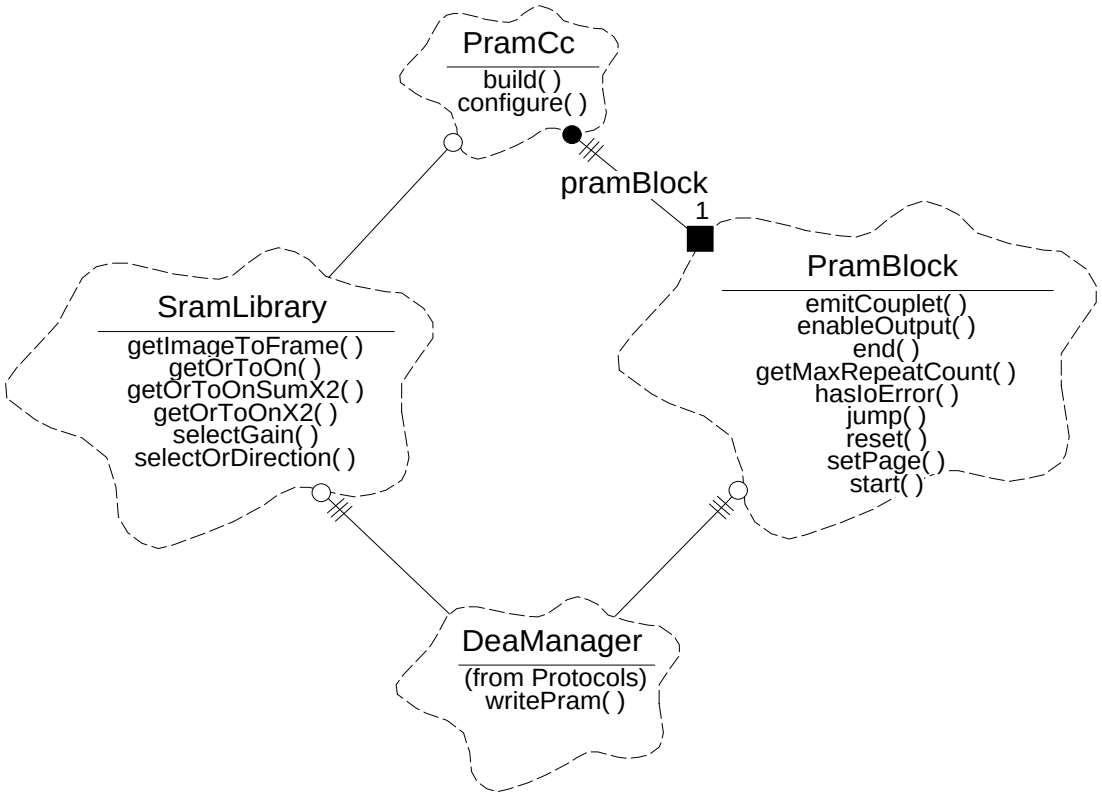
The following lists the uses of the Continuous Clocking PRAM Builder:

Use 1:: Build and load Program RAM instructions into a CCD-controller to perform Continuous Clocking mode CCD clocking.

35.3 Organization

The following illustrates the relationships used by the Continuous Clocking PRAM Builder class, **`PramCc`**. A detailed description of the **`PramBlock`** class is provided in Section 34.0.

FIGURE 171. Continuous Clocking PRAM Builder class relationships



`PramCc` - This class is responsible for generating and loading a series of sequencer Program RAM (PRAM) words needed to clock a CCD for Continuous Clocking mode. It provides a function which sets the desired clocking parameters (`configure`), and provides a function which generates and loads the clocking sequence into a particular DEA CCD-controller (`build`).

`PramBlock` - This class is responsible for emitting PRAM blocks to the DEA, each containing a header and one or more couplet entries. It is used by PRAM builders, such as **`PramTe`** (see Section 34.0) and **`PramCc`**, to load entries into PRAM. It provides functions which reset the instance to the start of PRAM and disable output to the DEA (`reset`), start and end a PRAM block (`start`, `end`), cause jump to a new PRAM page (`jump`), start writing to a new page within PRAM (`setPage`), emit a PRAM couplet (`emitCouplet`), obtain the maximum repeat cycles supported by a single couplet (`getMaxRepeatCount`), determine if an error was encountered while writing to the DEA (`hasIoError`), and enable output of PRAM words to the DEA (`enableOutput`). A complete description of this class is provided with the description of the Timed Exposure PRAM Builder classes in Section 34.0

`DeaManager` - This class is provided by the ***`Protocols`*** class category, and is responsible for managing access to the DEA interface. It provides a function which the builder uses to load words into a CCD-controller's Program RAM (`writePram`).

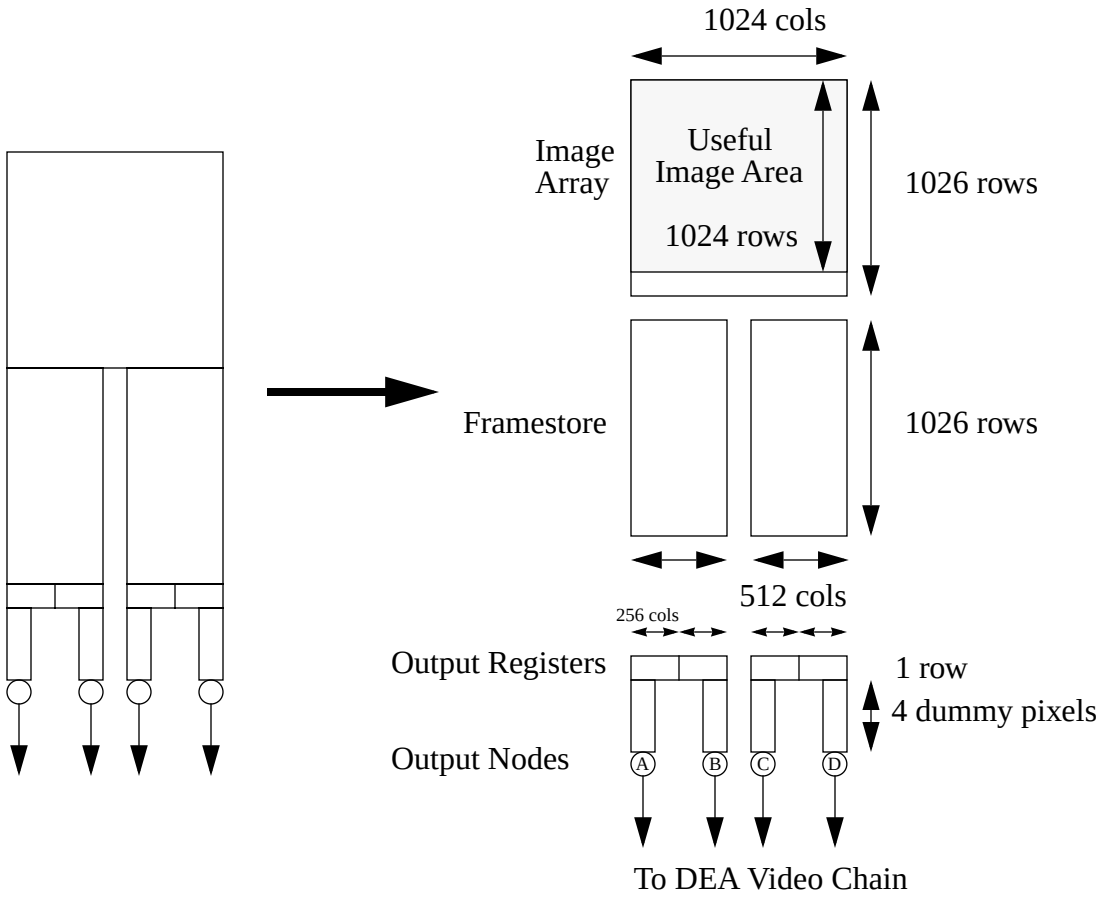
`SramLibrary` - This class is responsible for providing the PRAM builder with the Sequencer RAM (SRAM) locations of various primitives (NOTE: The function names abbreviate "output register" as "Or" and output node as "On"). It provides functions which select certain clocking options to use, such as the attenuated timing for the output register clocking, or the direction to clock the output registers (`selectGain`, `selectOrDirection`). It provides functions which supply the address of an SRAM block which clock and sample 1 pixel from the output register to the output node (`getOrToOn`), clock and sum two pixels from the output register to the output node (`getOrToOnX2`), clock and sum, without sampling, two pixels into the output register (`getOrToOnSumX2`). It also provides a function which returns the starting SRAM block and number of contiguous blocks used to clock one row from the image array to the framestore and from the framestore to the output registers (`getImageToFrame`).

35.4 PRAM Builder Design Issues

35.4.1 CCD Organization

Figure 172 illustrates a graphical representation of a single CCD. The figure on the left of the illustration presents a simplified picture of the main CCD components, and the figure on the right illustrates the pixel dimensions of each of the components.

FIGURE 172. Graphical CCD Representation

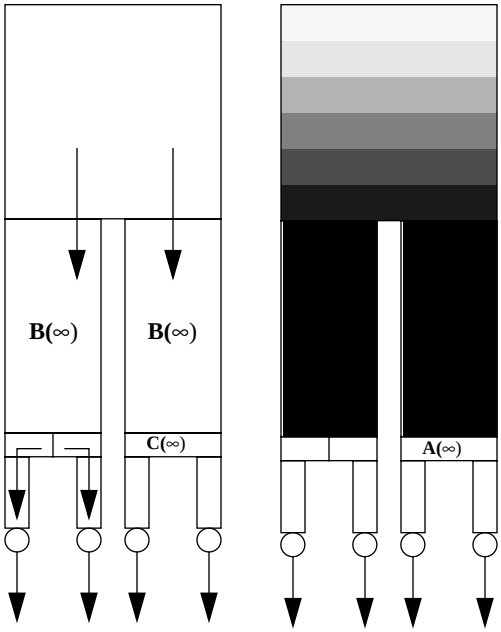


Each CCD consists of an Image Array, a Framestore, two output registers and four output nodes. Only 1024 of the 1026 Image Array rows are used for data acquisition.

35.4.2 Continuous Clocking Sequence

The following illustrates the sequence of operations used to clock out one summed CCD row in Continuous Clocking Mode (NOTE: For a description of the used SRAM primitives, see Section 35.4.3. The PRAM builder function associated with the action is posted at the bottom of each description):

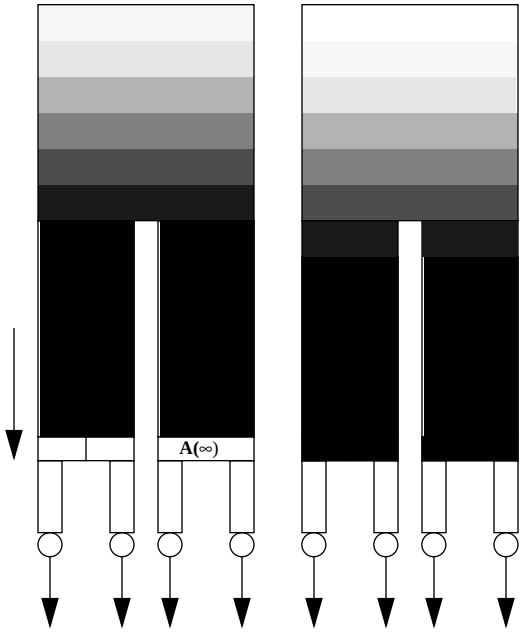
1. Discard all rows in image array and framestore



Repeatedly clock and discard 2052 CCD rows (1026 * 2), as described below by steps 2 through 5, to clear the contents of the image array and framestore, while establishing a constant integration time and noise profile of subsequent rows. The shading in the figure on the right is intended to represent the effective integration times of the rows within the CCD's image array. Although not shown in the figure, the exposure time of each row in the framestore to background radiation is a function of its position in the framestore. By the time a given row reaches the output register, the effective exposure to events and background is the same for rows preceding it.

(see `emitDataSet()`)

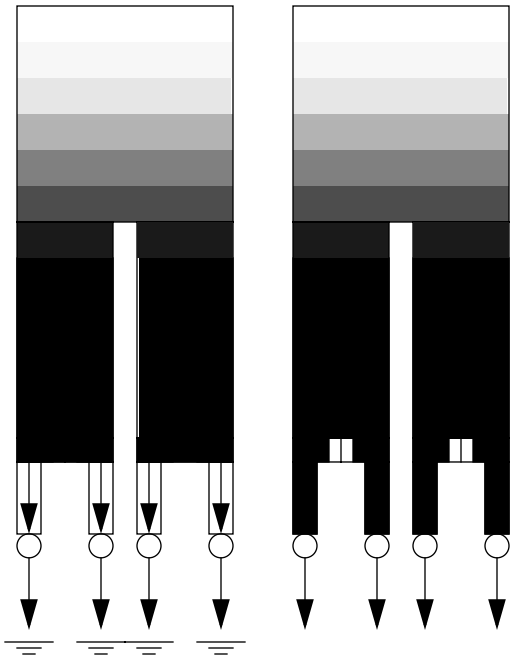
2. Clock and sum rows into output register



Concurrently, clock the desired number of rows from the image array into the framestore and from the framestore into the output registers. The rows clocked into the output register are summed.

(see `emitImageToFrame()`)

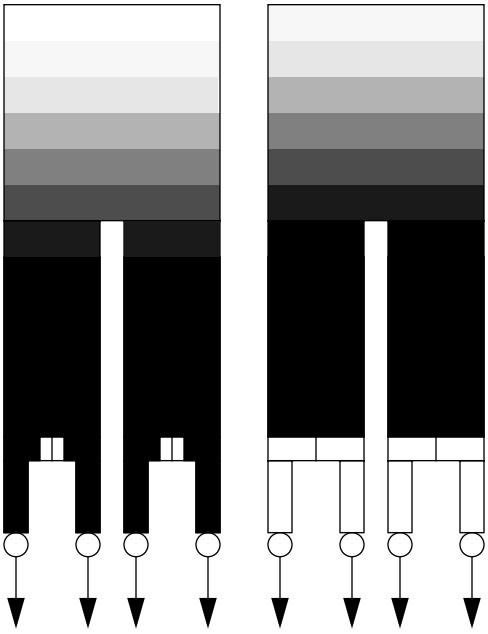
3. Clock and discard first four dummy columns



Clock four dummy column pixels from the output registers to the output nodes and discard the charge. On the last dummy pixel, emit a pixel code indicating the start of a data set.

(see `emitDiscardOr()`)

4. Clock row to DEA/DPA for processing

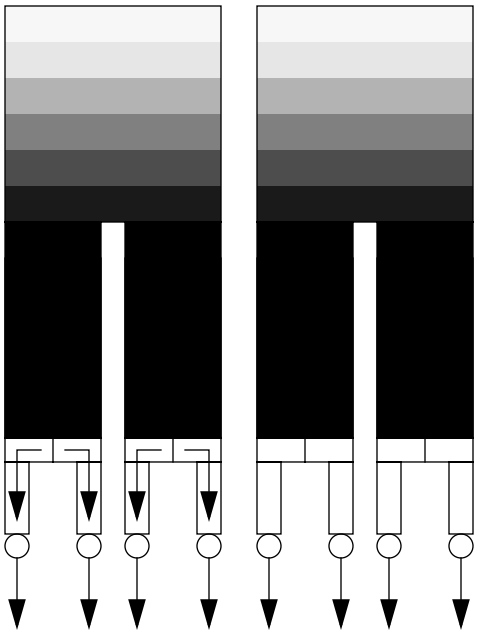


To DEA Video Chain

Clock 256 (512 if only using 2 output nodes) pixels from the output registers to the output nodes, sampling each and flagging the pixels to be processed. Meanwhile, the rows in the image array continue to integrate. NOTE: If performing on-chip summation, use an SRAM block which clocks and sums two pixels at a time to sum pairs of pixels at the output node. On the last pixel or pair of pixels, perform a final summation and sample the result.

(see `emitSummedPixel()`)

5. Overclock output registers

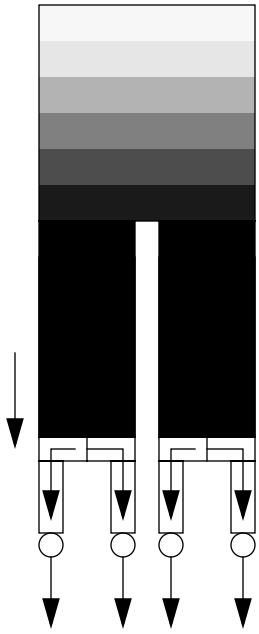


To DEA Video Chain

Clock the output registers a number of times, discarding the output, and then clock the output registers to produce the configured number of overclock pixels.

(see `emitDiscardOr()` and `emitSummedPixel()`)

6. Repeat 512 times to produce a complete data set



Repeat the steps starting from step 2 until an entire set of rows have been output. (NOTE: The number of rows output per data set is chosen to facilitate data processing and provide coarse timing markers. The number of rows does not affect the effective integration time of the rows, nor does it affect the continuous nature of the data). Continue producing data sets using steps 2 through 6 for the remainder of the science run.

(see `emitDataSet()`)

35.4.3 SRAM Primitives

SRAM consists of a collection of blocks, where each block performs a clocking operation. Some operations can be performed in a single SRAM block, whereas others, due to power constraints, require a series of SRAM blocks. Table 29 lists the SRAM operations used for Continuous Clocking Mode, where each SRAM block takes 1 pixel clock (~10μs) to execute.

TABLE 29. Continuous Clocking SRAM Operations

Operation	Number of SRAM Blocks	Description
Image to Frame	4	Clock one row from the image array to the framestore, and from the framestore to the output registers.
OR to ON discard (x2)	1	Clock two pixels from the output registers to their output nodes and discard the result.
OR to ON sum, x2	1	Clock and sum two pixels from the output registers into the output nodes.
OR to ON sample, standard	1	Clock one pixel from the output registers to their output nodes and sample the result (1 electron/ADU).
OR to ON sample, attenuated	1	Clock one pixel from the output registers to their output nodes and sample the result. Use signal timing to attenuate the gain (4 electrons/ADU).
OR to ON sample, x2, standard	1	Clock two pixels from the output registers to their output nodes, and sum and sample the pixels at the output nodes.
OR to ON sample, x2, attenuated	1	Clock two pixels from the output registers to their output nodes, and sum and sample the pixels at the output nodes.
Reverse OR to ON sum, x2	1	Clock and sum two pixels from the output registers away from their output nodes.[a]
Reverse OR to ON discard (x2 summing)	1	Clock two pixels from the output registers away from their output nodes and discard the result.
Reverse OR to ON sample, standard	1	Clock one pixel from the output registers away from their output nodes and sample the result.
Reverse OR to ON sample, attenuated	1	Clock one pixel from the output registers away from their output nodes and sample the result. Use signal timing to attenuate the gain (4 electrons/ADU).
Reverse OR to ON sample, x2, standard	1	Clock two pixels from the output registers away from their output nodes, and sum and sample the pixels at the output nodes.
Reverse OR to ON sample, x2, attenuated	1	Clock two pixels from the output registers away from their output nodes, and sum and sample the pixels at the output nodes.

a. The use of only two output nodes, such as in AC and BD mode, is selected via a discrete command to the DEA CCD controller to swap the output register phase clocks.

35.4.4 PRAM Headers and Couplets

This section describes the format of the PRAM words. PRAM consists of collections of word pairs known as ‘couplets.’ Each couplet contains the address of one SRAM block to invoke, a repeat counter indicating the number of times to repeat the block, and a pixel code, which instructs the Front End Processors (FEPs) how to process pixels being clocked while the SRAM block is being invoked. Each invocation of a couplet takes one pixel clock period (~10μs).

While PRAM is running, pixel clocks are always being delivered to the FEPs, regardless of whether pixel data is being clocked out of the CCDs or not. The pixel code portion of the couplets indicate how the FEPs interpret the data. Valid pixel data is flagged using a “Valid Pixel” code, and overclock pixels are flagged using an “Overclock” code. All other codes cause the clocked pixel data to be ignored. These include the “Vsync” code, which indicates that a new image has been started, the “Hsync” code, which indicates that a new image row is about arrive, and the “Ignore” code, which indicates that the clocked pixel should be completely ignored by the FEPs.

Each collection of couplets is preceded by a pair of ‘header’ words. The header consists of the number of couplets which follow the block, the number of times to repeat the collection of couplets, and an action to perform once the collection of couplets. These actions include: halt the sequencer, continue to the header immediately following the couplet collection, jump to the first PRAM location (restart), and jump to the indicated PRAM page.

The format of this header is as follows:

15	14	13 12	11 0
1	1	Option	PRAM Block Repeat Count
1	0	Page Jump	Couplet Count

PRAM Block Repeat Count.....This specifies the number of times to repeat the entire block minus 1

Option.....This specifies the next sequence option [0: Restart, 1:Continue, 2:Halt, 3:Page Jump]

Couplet Count .....This specifies the number of PRAM word pairs (couplets) following the block minus 1

Page Jump .....If Option is 3, this specifies the PRAM page to jump to

The following illustrates the format of the PRAM couplets within a block:

15	14	13 12	11 5	4	3 0
0	1	SRAM Page Address		0	PixCode
0	0	0	0	Major Cycle Count	

PixCode.....This code is sent to the Front End Processor with each major cycle (pixel), [0:Ignore, 3:Valid Pixel, 4:End of Row (HSYNCH), 8:Start of Image (VSYNCH), 12:Overclock]

SRAM Page Address.....This specifies which block of 64 SRAM blocks should be sequenced during a major cycle.

Major Cycle Count.....This specifies how many times to repeat the selected SRAM block minus 1

PRAM Memory is organized around four contiguous pages, each containing 8192 words. The “jump” option of the PRAM Header causes PRAM to transfer control to the beginning of the specified page once the current PRAM block completes execution.

35.4.5 Clocking Algorithm

The following illustrates the clocking algorithm using pseudo-code, where the **bolded** text indicates operations which utilize the header repeat counter, *italicized* text indicates operations which can be accomplished using a single SRAM block and utilize the couplet repeat counter, and the underlined text indicates looping operations which are "unrolled" by the builder:

```
PRAM Page 0:
    /* ---- Clock and emit first row of data set ---- */
    Clock image array/framestore to sum "n" rows in output registers
    Discard 4 dummy pixels, and indicate start of data set (VSYNC)
    REPEAT for all summed pixels in output register
        Clock output register to sum "m" pixels. Sample result.
    ENDREPEAT
    Discard output register pixels prior to producing overclocks
    REPEAT for each overclock pixel sum
        Clock output register to sum "m" overclocks. Sample result.
    ENDREPEAT
    Indicate end of row (HSYNC)

    /* ---- Clock remaining rows of data set ---- */
    REPEAT for 511 rows
        Clock image array/framestore to sum "n" rows in output registers
        Discard dummy pixels
        REPEAT for all summed pixels in output register
            Clock output register to sum "m" pixels. Sample result.
        ENDREPEAT
        Discard output register pixels prior to producing overclocks
        REPEAT for each overclock pixel sum
            Clock output register to sum "m" overclocks. Sample result.
        ENDREPEAT
        Indicate end of row (HSYNC)
    ENDREPEAT
    Jump to PRAM Page 0
```

35.4.6 Parallel Transfers with Multiple CCDs

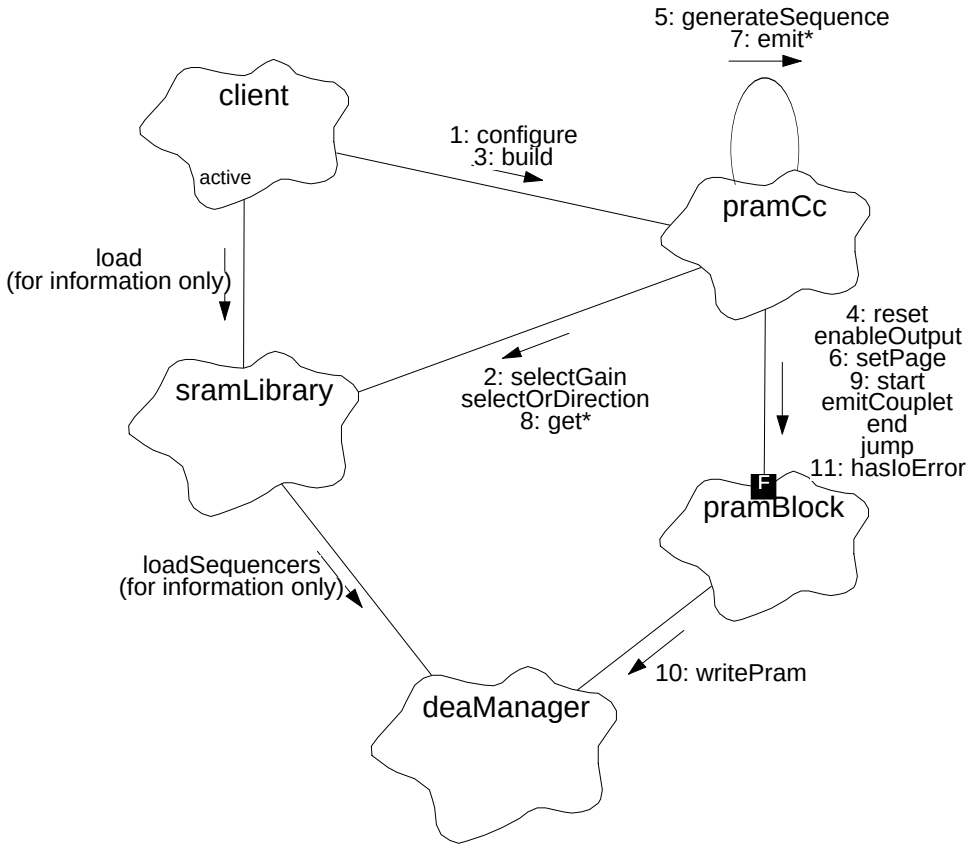
Assuming constraints on the maximum number of rows and columns that may be summed, phase-delays between image array/framestore row transfers on different CCDs, such as those used by Timed Exposure Mode (see Section 34.4.6), are not required.

35.5 Scenarios

35.5.1 Use 1: Build and load PRAM to perform Continuous CCD Clocking

Figure 173 illustrates the use of the **`PramCc`** class to generate a PRAM load for Continuous Clocking mode (NOTE: The `load` call to *`sramLibrary`*, and `loadSequencers` call to *`deaManager`* are for reference information only. For more detail, see Section 36.0). For a description of the use of the **`PramBlock`** class, see Section 34.5.1

FIGURE 173. Build PRAM Load

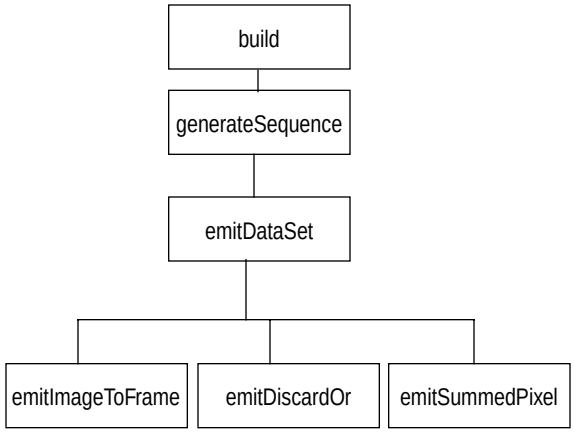


1. The *client* configures the builder, passing the clocking parameters, and pointer to the *sramLibrary* to *pramCc*.configure().
2. configure() initializes the *pramCc*'s instance variables and then selects which gain and output register clocking direction primitives to use, using *sramLibrary*.selectGain() and *sramLibrary*.selectOrDirection(), respectively.
3. The *client* then passes the CCD Id to *pramCc*.build() to generate the clocking sequence, and load the sequence into the DEA.
4. build() first invokes *pramBlock*.reset() reset *pramBlock*'s instance variables, and then calls pramBlock.*enableOutput*() to enable output to the DEA.
5. build() then calls generateSequence() to generate the clocking sequence and load it into the DEA.
6. generateSequence() uses *pramBlock*.setPage() to set the initial PRAM page to write to.
7. generateSequence() uses a variety of *pramCc*.emit*() functions to emit the various phases of the clocking sequence (see Figure 174 for a call structure chart).
8. The emit*() functions use the various *sramLibrary*.get*() functions to obtain the address and block count of the SRAM primitives used to perform certain functions.

9. The `emit*`() functions use *`pramBlock`*`.start`()/`emitCouplet`()/`end`()/`jump`() functions to generate the PRAM blocks (i.e. header followed by one or more couplets) needed to clock the CCD (see Section 34.5.1for the use of the **`PramBlock`** class).
10. The *`pramBlock`* functions use *`deaManager`*`.writePram`() to load the PRAM words into the DEA.
11. If an error is encountered during a write, *`pramBlock`* sets an internal I/O error flag and disables further writes, but allows the sequence generation to complete. Once *`pramCc`*`.generateSequence`() returns, the caller, *`pramCc`*`.build`(), then examines whether or not an I/O error was encountered using *`pramBlock`*`.hasIoError`(), and reports the condition in its return to the calling client.

Figure 174 illustrates a structure chart, indicating the functional hierarchy within the **`PramTe`** class:

FIGURE 174. PramCc Class Internal Structure Chart



35.6 Class PramCc

Documentation:

This class generates the Program RAM sequence needed to perform Continuous CCD Clocking.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

SramLibrary

Public Interface:

Operations:
`PramCc()`
`build()`
`configure()`

Protected Interface:

Operations:
`emitDataSet()`
`emitDiscardOr()`
`emitImageToFrame()`
`emitSummedPixel()`
`generateSequence()`

Private Interface:

Has-A Relationships:

unsigned *`rowSum`*: Number of rows to sum on-chip.This is copied by `configure`().

unsigned *`colSum`*: Number of columns to sum on-chip. This is copied by `configure`().

`QuadMode` *`qMode`*: Quadrant Configuration: Full, Diagnostic, AC, or BD. This is copied by `configure`().

unsigned *`overclockPairs`*: Number of pairs of overclock pixels to output. This is copied by `configure`().

Boolean *gain4*: Determines whether or not to use high-gain SRAM timing. If BoolFalse, use 1:1 SRAM, if BoolTrue, use 1:4 SRAM timing. This is copied by `configure`().

const unsigned *arrayRows*: Number of non-summed rows in the CCD Image and Frame Store

const unsigned *nodeCols*: Number of non-summed columns per output node (assuming all 4 nodes in use)

const unsigned *dummyCols*: Number of dummy columns between the output register and the output node.

const unsigned *initialRows*: Number of rows to initially discard at beginning of run to achieve consistent integration times.

const unsigned *ocDummy*: This contains the number of dummy pixels to clock out of the output register prior to clocking out overclock pixels.

const unsigned *summedRows*: This is the total number of summed rows to clock out of the CCD per data set.

SramLibrary* *sramLibrary*: This points to the SRAM library to use when loading PRAM.

PramBlock *pramBlock*: This is used by the Continuous Clocking PRAM Builder to emit blocks of PRAM couplets.

unsigned *shiftCols*: This specifies the number of unsummed columns per output node (not including dummy columns).

unsigned *summedShiftCols*: This specifies the number of summed columns to shift out for each row.

Boolean *configured*: This flag indicates whether or not `configure`() has been called. If not, this field contains *BoolFalse*. If `configure`() has been called the field is *BoolTrue*.

Concurrency: Guarded

Persistence: Transient

35.6.1 PramCc()

Public member of: **PramCc**

Documentation:

This function is the constructor for the **PramCc** class. This function initializes the class instance variables, and sets *configured* to *BoolFalse*.

Concurrency: Guarded

35.6.2 build()

Public member of: **PramCc**

Return Class: **Boolean**

Arguments:

CcdId *ccdId*

Documentation:

This function builds a PRAM sequence and loads the sequence into the DEA CCD-controller specified by *ccdId*. The function returns *BoolTrue* if the sequence is successfully built and loaded into the DEA. It returns *BoolFalse* if an error was encountered while trying to write to the DEA.

Preconditions:

configure() must have been called (*configured* == *BoolTrue*).

Semantics.

Call *pramBlock*.reset() and then pass *ccdId* to *pramBlock*.enableOutput(). Build and load the sequence using generateSequence(), and check for I/O errors using *pramBlock*.hasIoError(). If an error was encountered, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Guarded

35.6.3 configure()

Public member of: **PramCc**

Return Class: **Boolean**

Arguments:

unsigned *rowsum*
unsigned *colsum*
QuadMode *quadrants*
unsigned *ocPairs*
Boolean *attenuate*
SramLibrary& *sramlib*

Documentation:

This function configures the Continuous Clocking PRAM builder. *rowsum* is the number of rows to sum, and *colsum* is the number of columns to sum per output pixel. *quadrants* indicates the output node configuration and clocking direction (Full, Diagnostic, AC, or BD). *ocPairs* indicates the number of pairs of overclock pixels to generate. *attenuate* selects the SRAM timing. If *BoolFalse*, use 1 electron/ADU timing, and if *BoolTrue*, use timing which produces 4 electrons/ADU. *sramlib* is a reference to the SRAM library to use when building PRAM.

Preconditions:

rowsum and *colsum* must be less than some TBD values (determined by TBD DEA power constraints).

Semantics.

Copy the parameters into the corresponding instance variables, configure the SRAM library's gain and output register direction, and compute derived clocking count information. Finally, set *configured* to *BoolTrue*.

Concurrency: Guarded

35.6.4 emitDataSet()

Protected member of: **PramCc**

Return Class: **Boolean**

Arguments:

unsigned *nrows*
Boolean *flush*

Documentation:

Emit PRAM instructions to produce 1 data set of continuously clocked CCD pixels. *nrows* specifies the number of rows to emit in the data set, and *flush* indicates whether or not to use the produced data set. If *flush* is *BoolFalse*, treat clocked data as valid pixels. If *flush* is *BoolTrue*, then ignore all clocked data. This is used to discard the initial contents of the image array and framestore at the start of the run.

Semantics.

Use two passes to produce the PRAM load for a data set. On the first pass, emit the PRAM instructions to sum and transfer the first row of the data set. The second pass emits a PRAM loop which transfers the remaining rows of the data set.

On each pass, call `emitImageToFrame`() to clock *rowSum* rows from the image array to the framestore, and from the framestore into the output registers. The rows are summed into the output registers. Then call `emitDiscardOr`() to discard the dummy pixels from the output register. On the first pass, flag the last dummy pixel with a VSYNC pixel code, to indicate the start of the data set. Once the dummy pixels are discarded, call `emitSummedPixel`() to sample *summedShiftCols* summed output pixels from output register. If overclocks are being used, call `emitDiscardOr`() to discard a number of pixels prior to clocking the overclocks, and then call `emitSummedPixel`() to sample (2**overclockPairs*) summed overclock pixel values. Delimit the end of each row by passing HSYNC to `emitDiscardOr`().

Concurrency: Guarded

35.6.5 emitDiscardOr()

Protected member of: **PramCc**

Return Class: **Boolean**

Arguments:

unsigned *npix*
unsigned *pixcode*

Documentation:

Emit PRAM couplets which clock *npix* columns from the output register into the output node, each time, discharging the output node, effectively discarding the pixels. *pixcode* indicates the pixel code to use when discarding the pixels.

Semantics.

If *npix* is greater than or equal to 2, start by discarding pixels two at a time by fetching the SRAM block using *sramLibrary*->getOrToOnX2(), and emitting the couplet with a repeat count of *npix*/2, using *pramBlock*.emitCouplet(). Pass *pixcode* as the pixel code to emit while performing the clocking. Then adjust *npix* to contain the remainder (NOTE: Use of a remainder is a hook in case faster SRAM blocks become available).

If *npix* is not zero (in this case, it will be 1), get the SRAM block to discard 1 pixel using *sramLibrary*->getOrToOn(), and emit the couplet to repeat *npix* times, using *pramBlock*.emitCouplet() and passing *pixcode* as the pixel code to emit while performing the clocking.

Concurrency: Guarded

35.6.6 emitImageToFrame()

Protected member of: **PramCc**

Return Class: **Boolean**

Arguments:

unsigned *nrows*

Documentation:

This function clocks *nrows* from the image array to the framestore, and from the framestore into the output register. The rows clocked into the output register are summed with each other, and with charge already in the output registers.

Semantics.

Call *sramLibrary*->`getImageToFrame`() to obtain the starting index of the series of blocks which transfer one row from the image array to the framestore and from the framestore to the output register, and to obtain the number of adjacent blocks need for the transfer. If only 1 block is required, call *pramBlock*.`emitCouplet`(), passing the index of the block, and a repeat count of *nrows* to emit the PRAM instructions for the transfer.

If more than one block is needed and more than one row is being output, emit the series of blocks *nrows* times (NOTE: In this clocking mode the repeat block is used to count summed output rows, rather than rows being summed).

Concurrency: Guarded

35.6.7 emitSummedPixel()

Protected member of: **PramCc**

Return Class: **Boolean**

Arguments:

unsigned *npix*
PRAM_PIXCODE *pixcode*

Documentation:

Emit PRAM couplets which emit a series of *npix* summed pixels from the output register. Each output pixel is flagged with the *pixcode* pixel code, and consists of the sum of *colSum* output register pixels.

Semantics.

This function starts by obtaining the SRAM indices which sum and accumulate a pair of pixels, sum a pair of pixels and sample the accumulated value, and add a single pixel and sample the accumulated result, using *sramLibrary*->getOrToOrSumX2(), *sramLibrary*->getOrToOnX2() and *sramLibrary*->getOrToOn(), respectively.

It then enters a loop which iterates *npix* times. Each iteration sums *colSum* pixels and emits a couplet specifying an SRAM block using *pramBlock*.emitCouplet(). The SRAM blocks are chosen to minimize the number of cycles needed to perform the operation for each pixel. If *colSum* is greater than two, it emits a couplet which sums and accumulates pairs of pixels *colSum*/2 times. If *colSum* is even, it emits a final couplet which sums two more pixels and samples the result. However, if *colSum* is odd, it emits a final couplet which accumulates just one more pixel and samples the accumulated sum.

Concurrency: Guarded

35.6.8 generateSequence()

Protected member of: **PramCc**

Return Class: **Boolean**

Documentation:

This function runs through the PRAM generation process. The first portion of the function emits PRAM instructions which flush the initial contents of the image array, and the second portion calls `emitDataSet`() to emit PRAM instructions which produce 1 data set worth of continuous clocking pixels.

Semantics.

Generate the main clocking sequence in PRAM Page 0. Call *`pramBlock`*`.setPage`() to start writing to PRAM Page 0. Then call `emitDataSet`() to produce the PRAM load which clocks a single data set of 512 summed rows. Then call *`pramBlock`*`.jump`() to loop back to the start of PRAM Page 0 to repeatedly produce data sets.

Concurrency: Guarded

36.0 Sram Library (36-53241 A)

36.1 Purpose

The SRAM Library provides a set of primitives which direct CCD pixel charge manipulation in support of specific predetermined observational goals.

36.2 Uses

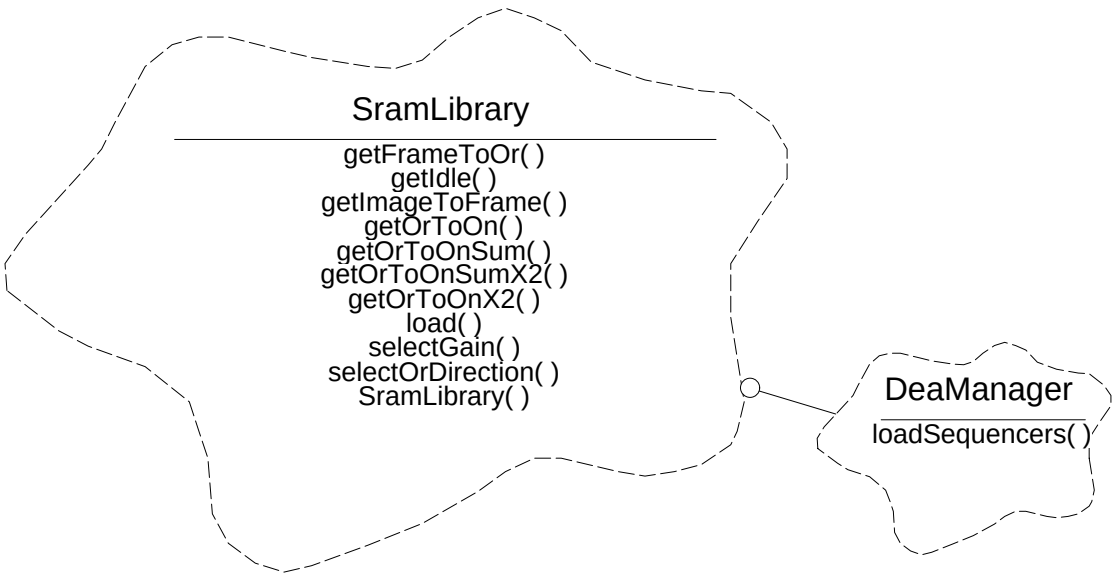
The following lists the uses of the SRAM Library.

Use 1:: Facilitates loading of the SRAM Library by the DEA Manager
Use 2:: Accepts and retains a set of decision directives
Use 3:: Directs access to one or more blocks containing a primitive command set

36.3 Organization

Figure 175 illustrates the relationship used by the Sram Library.

FIGURE 175. Sram Library Relationships



The Sram Library class provides accesses to a catalog of primatives. The primitives provide a set of directives which are implemented by the CCD controller to initiate each step of each action of a series of exposures which make up an observation. These are the directives which cause the shifting of CCD pixel charge out of the CCD for processing. This class uses the services of the DEA Manager to load the library.

36.4 SRAM Load Table Format

This section illustrates the format of an SRAM library table, used to load SRAM and determine where the various SRAM primitives reside once loaded.

```
struct SramEntry
{
    unsigned  index;                          // Location
    unsigned  count;                          // Blocks
};

struct SramHeader // Catalog of entries     accessor function
{
    struct SramEntry idle;                    // getIdle

    struct SramEntry imageFrame;              // getImageToFrame
    struct SramEntry frameOReg;               // get FrameToOr

    struct SramEntry oRegONode;               // getOrToOn
    struct SramEntry oRegONode_High;          // getOrToOn
    struct SramEntry oRegONodeX2;             // getOrToOnX2
    struct SramEntry oRegONodeX2_High;        // getOrToOnX2
    struct SramEntry oRegONode_Rev;           // getOrToOn
    struct SramEntry oRegONode_High_Rev;      // getOrToOn
    struct SramEntry oRegONodeX2_Rev;         // getOrToOnX2
    struct SramEntry oRegONodeX2_High_Rev;    // getOrToOnX2

    struct SramEntry oRegONode_Dis;           // getOrToOn
    struct SramEntry oRegONodeX2_Dis;         // getOrToOnX2
    struct SramEntry oRegONode_Dis_Rev;       // getOrToOn
    struct SramEntry oRegONodeX2_Dis_Rev;     // getOrToOnX2

    struct SramEntry oRegONodeSum;            // getOrToOnSum
    struct SramEntry oRegONodeSum_High;       // getOrToOnSum
    struct SramEntry oRegONodeSum_Rev;        // getOrToOnSum
    struct SramEntry oRegONodeSum_High_Rev;   // getOrToOnSum

    struct SramEntry oRegONodeSumX2;          // getOrToOnSumX2
    struct SramEntry oRegONodeSumX2_Rev;      // getOrToOnSumX2
};

struct SramTable
{
    struct SramHeader sramHeader;
    struct DeaSequenceLoad seqLdBlock;        // from cmdif.H
};
```

36.5 Scenario

The library itself consists of a catalog of the CCD control primitives and the primitive sets themselves. When setting up an observation, the selected Sram Library address is provided to the constructor, `SramLibrary`(). The catalog structure delineates the location of each of the primatives. Each primitive is contained in one or more contiguous data blocks. The library functions, in conjunction with the decision directives, are used in determining the location and length of the primitives which will control each step of an observation in a science run.

Use 1:

The client uses `load`() which directs the **`DeaManager`**`.loadSequencers`() to load the sets of primitives into preselected memory locations. The catalog, having been properly constructed to indicate the location of the loaded primitives, will be called upon to direct the client to the desired primitive for sequencing of the CCDs.

Use 2:

The two functions, `selectGain`() and `selectOrDirection`() provide boolean decision indicators which enable four functions to select and deliver each of eighteen primitives. The default settings are: low gain and forward direction.

Use 3:

The function names are indicative of the primitive they provide. Or and On are output register and output node. X2 indicates a shift of charge from two pixels (default is one pixel). Sum denotes a accumulation of charge at the node. Image to Frame and Frame to Output Register refer to transfer of a row of pixel charge. Idle provides for CCD exposure time.

The client will access the catalog functions to obtain the location and block count of the primitives to be invoked in controlling the CCD output.

36.6 Class SramLibrary

Documentation:

This class supplies a collection of SRAM blocks which perform specific operations. The member functions of this class return the index and block count of an SRAM primitive which performs a particular operation.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

DeaManager

Public Interface:

Operations:

```
SramLibrary()
getFrameToOr()
getIdle()
getImageToFrame()
getOrToOn()
getOrToOnSum()
getOrToOnSumX2()
getOrToOnX2()
load()
selectGain()
selectOrDirection()
```

Private Interface:

Has-A Relationships:

Boolean *useHighGain*: This specifies whether or not to use high-gain OR to ON clocking. If BoolFalse, use the 1:1 gain selection. If BoolTrue, use the 4:1 gain selection.

Boolean *useReverse*: This specifies the direction in which to clock the output registers. If BoolFalse, clock in the normal clocking direction (i.e. Full or AC). If BoolTrue, clock in the reverse direction (BD or Diagnostic).

const SramTable* *catalog*: This points to the table passed as the argument to the instance's constructor.

Concurrency: Sequential

Persistence: Transient

36.6.1 SramLibrary()

Public member of: **SramLibrary**

Arguments:

SramTable* *catalogAddr*

Documentation:

SramLibrary This constructor initializes the SramLibrary instance, where *catalogAddr* points to the library table to use.

Concurrency: Sequential

36.6.2 getFrameToOr()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

unsigned& *index*
unsigned& *count*

Documentation:

This function supplies a set of adjacent SRAM blocks which clock one row from the framestore into the output registers. Upon return, *index* contain the index of the first SRAM block, and *count* returns the number of adjacent blocks to use.

Concurrency: Guarded

36.6.3 getIdle()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

unsigned& *index*

Documentation:

This function returns the *index* of an SRAM block which is used to integrate the CCDs.

Concurrency: Guarded

36.6.4 getImageToFrame()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

unsigned& *index*
unsigned& *count*

Documentation:

getImageToFrame will get a set of adjacent SRAM blocks which transfer one row from the image array to the framestore. On return, *index* contains the SRAM index to the first block, and *count* returns the number of adjacent blocks to use.

Concurrency: Guarded

36.6.5 getOrToOn()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

Boolean *discard*
unsigned& *index*

Documentation:

getOrToOn will get an SRAM block which transfers one pixel from the output register to the output node. If *discard* is BoolFalse, the caller intends to sample the pixels at the output node, and if *discard* is BoolTrue, the caller intends to discard the pixel. Upon return, *index* will contain the SRAM index of the block. The supplied block will clock the output registers in the direction indicated by the most recent call to `selectDirection`() and with the gain specified by the most recent call to `selectGain`().

Concurrency: Guarded

36.6.6 getOrToOnSum()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

unsigned& *index*

Documentation:

This function supplies an SRAM block which clocks and sums one pixel from the output register into the output node. Upon return, *index* contains the selected SRAM block. The generated block will use the gain and direction last selected by `selectGain`() and `selectOrDirection`(), respectively.

Concurrency: Guarded

36.6.7 getOrToOnSumX2()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

unsigned& *index*

Documentation:

This function returns an SRAM block which clocks and sums two output register pixels into the output node. Upon returning, *index* contains the SRAM address of the desired block.

Concurrency: Guarded

36.6.8 getOrToOnX2()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

Boolean *discard*
unsigned& *index*

Documentation:

This function supplies an SRAM block which clocks two pixels from the output register to the output node (possibly in one SRAM cycle). *discard* indicates that the caller wants the clocked pixels to be discarded (either at the output node, or via the FEP). Upon return, *index* contains the SRAM index of the desired block. The supplied block will clock in the direction last specified using `selectDirection`(), and is timed to generate the gain last chosen using `sellectGain`().

Concurrency: Guarded

36.6.9 load()

Public member of: **SramLibrary**

Return Class: **Boolean**

Documentation:

This function loads the contents of the SRAM library into the DEA controllers selected by the library's load block, using the DEA Manager. If the load succeeds the function returns BoolTrue, otherwise, it returns BoolFalse.

Concurrency: Guarded

36.6.10 selectGain()

Public member of: **SramLibrary**

Return Class: **void**

Arguments:

Boolean *high*

Documentation:

The selectGain function selects whether or not to use SRAM primitives which return high gain from the output registers. If high is BoolFalse, then use primitives which supply a gain of 1:1. If high is BoolTrue, return use blocks which supply a gain of 4:1 at the output node (sacrificing pulse-height resolution).

Concurrency: Guarded

36.6.11 selectOrDirection()

Public member of: **SramLibrary**

Return Class: **`void`**

Arguments:

`Boolean` *`reverse`*

Documentation:

The `selectOrDirection` function selects the clocking direction of the output registers. If *`reverse`* is BoolFalse, then clock each register half toward their respective output nodes (i.e. use for Full and AC modes). If *`reverse`* is BoolTrue, clock each register in the opposite direction (i.e. use for BD and Diagnostic modes).

Concurrency: Guarded

37.0 Science Data Processing Classes (36-53228 B)

37.1 Purpose

The purpose of the Science Processing classes is to implement the detailed science data processing requirements of the Back End Processor. The overall control of the data processing is managed by the Science Management Classes, described in Section 33.0.

37.2 Uses

The following lists the uses of the Science Processing classes:

Use 1:: Timed Exposure Raw Mode Data Processing
Use 2:: Timed Exposure Histogram Mode Data Processing
Use 3:: Timed Exposure Faint Mode 3x3 Event Data Processing
Use 4:: Timed Exposure Faint Mode 3x3 with Bias Data Processing
Use 5:: Timed Exposure Graded Mode Data Processing
Use 6:: Continuous Clocking Raw Mode Data Processing
Use 7:: Continuous Clocking Faint Mode 1x3 Event Data Processing
Use 8:: Continuous Clocking Graded Mode Data Processing

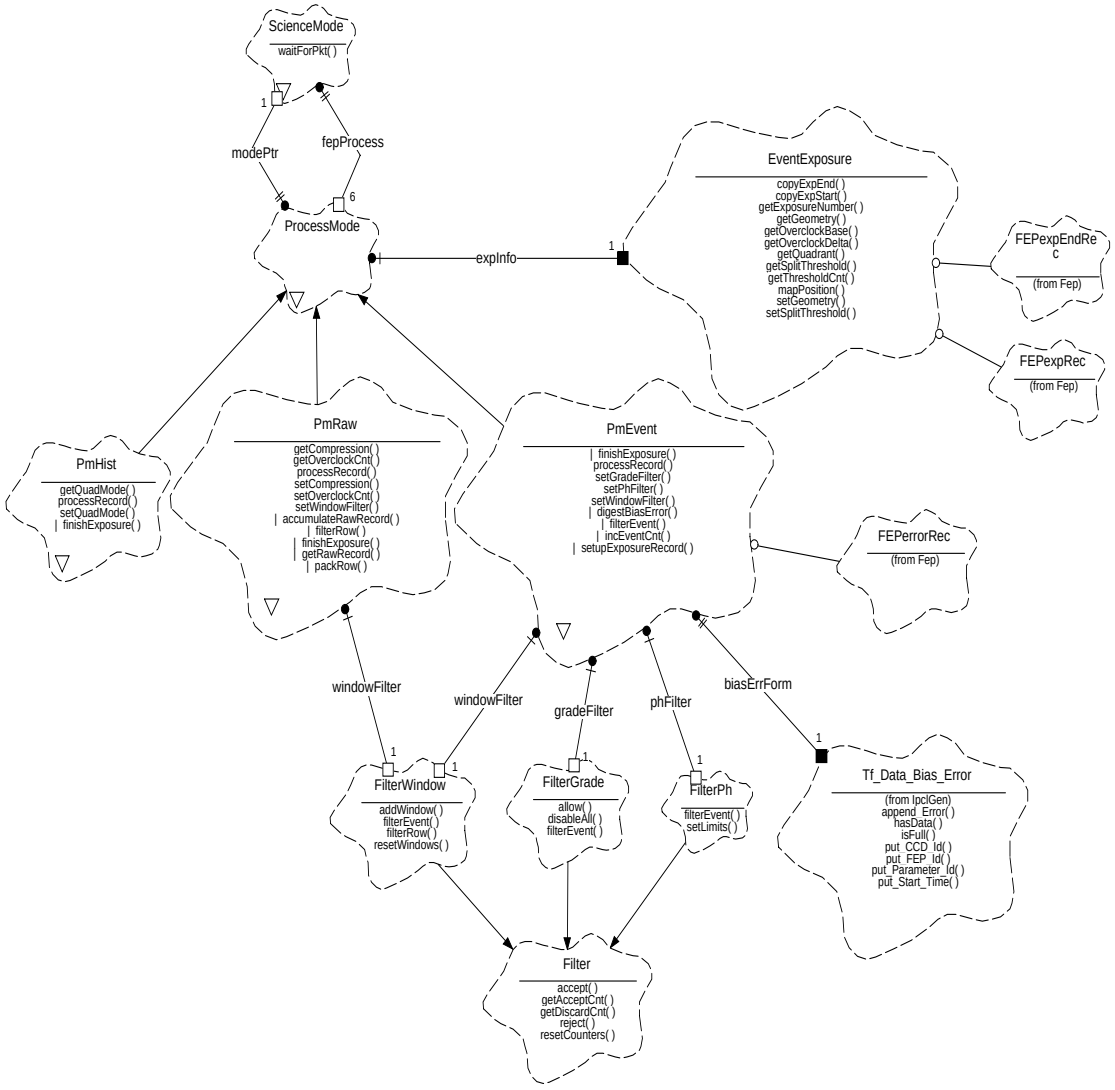
37.3 Organization

This section describes the class hierarchy and relationships between the various pieces of the science processing system. Given the large number of relationships between the various processing classes, this section is divided into subsections. Each subsection describes a portion of the system, focusing on a particular group of relationships. The first subsection establishes the abstract processing mode base class definitions, and the data filter class definitions. The second subsection describes the leaf processing mode classes, each of which is responsible for handling a single science data processing mode. The third subsection describes the data representation classes used within the system, and the last subsection describes how the Science Mode classes, **`SmTimedExposure`** and **`SmContClocking`**, relate to the processing mode classes.

37.3.1 Process Mode Base Classes

Figure 176 illustrates the class relationships of the processing mode base classes.

FIGURE 176. Processing Mode Base Class Relationships



`ScienceMode` - This class is an abstract base class representing a main science mode. It contains an array of pointers to **`ProcessMode`** instances, one pointer for each Front End Processor (FEP) in the instrument, and uses the **`ProcessMode`** instances to handle data produced by a given FEP. The **`ProcessMode`** class, in turn, contains a pointer back to a **`ScienceMode`** instance. **`ProcessMode`** uses the **`ScienceMode`** to wait for telemetry packet buffers to use when sending CCD science data and exposure records (`waitForPkt`). See Section 33.0 for a detailed description of the **`ScienceMode`** class. See Section 37.3.4 for descriptions of the specific science modes supported by the instrument.

ProcessMode - This class is an abstract class responsible for processing science data being emitted by a single Front End Processor. This abstract class implements functions common to all science processing modes, and defines interfaces required by all such modes. This class is described in detail in Section 33.0.

EventExposure - This class is responsible for maintaining science run and exposure information needed for data processing. It provides functions which store and retrieve the science run's row and column scale factors, the row offset, and the output node configuration used by images produced by a FEP (`setGeometry`, `getGeometry`). It has functions to set and get the split threshold levels configured for the science run (`setSplitThreshold`, `getSplitThreshold`). This class provides utility functions which use the reported run information, such as map a clocked pixel position into absolute CCD coordinates (`mapPosition`), and determine from which CCD quadrant a pixel was produced (`getQuadrant`). This class obtains its exposure information from records produced by a Front End Processor, **FEPexpRec** and **FEPexpEndRec** (`copyExpStart`, `copyExpEnd`). The extracted information is used to provide clients with the last reported exposure number (`getExposureNumber`), the reported overclock base values and delta values (`getOverclockBase`, `getOverclockDelta`), and the number of pixels detected by the FEP which were above their threshold levels (`getThresholdCnt`). Refer to Section 4.10 for a description of the **FEPexpRec** and **FEPexpEndRec** structure definitions.

PmHist - This class is a subclass of **ProcessMode**, and is responsible for processing histogram data produced by a given Front End Processor. This class provides functions which configure and query which video chains (quadrants) are being histogrammed (`setQuadMode`, `getQuadMode`), and which process FEP to BEP records (`processRecord`). It also defines an abstract function which subclasses must re-define to complete the current exposure (`finishExposure`).

PmRaw - This class is a subclass of **ProcessMode**, and is responsible for processing raw pixel data produced by a given Front End Processor. This class contains a reference to a **FilterWindow** class instance, which it uses to clip raw images produced by the FEP. This class provides functions which set and retrieve the compression table code (`setCompression`, `getCompression`), which set and query the number of overclock pixels in each raw row of data (`setOverclockCnt`, `getOverclockCnt`), which install a the window list to use for clipping the raw data (`setWindowFilter`), and which interpret data records produced by a FEP (`processRecord`). It also provides member functions used by its child classes to accumulate a series of FEP to BEP ring-buffer records into a single FEP to BEP record (`accumulateRawRecord`), retrieve a pointer to the accumulated record (`getRawRecord`), run the raw row through the window clipping filter (`filterRow`), and pack the unclipped regions of the row into an output buffer (`packRow`). It also defines an abstract function which subclasses must re-define to complete the current exposure (`finishExposure`).

PmEvent - This class is a subclass of **ProcessMode**, and is responsible for filtering candidate events produced by a given Front End Processor, for processing bias map errors reported by a FEP, and for processing FEP-supplied event-mode exposure record informa-

tion. This class contains three filter references. The **FilterWindow** instance is used to filter events based on their position within the CCD image. The **FilterGrade** instance is used to filter events based on the spatial distribution of pulse heights within the pixels representing the event. The **FilterPh** instance is used to filter events based on their pulse height. The **PmEvent** class provides functions which are used during setup to establish the pointers to these filters (`setWindowFilter`, `setGradeFilter`, `setPhFilter`). It provides functions which interpret data records produced by a FEP (`processRecord`), and process bias error records (`digestBiasError`). It also provides functions used by its child classes to setup exposure record packets (`setupExposureRecord`). It also provides functions which filter out events using its event filters (`filterEvent`), and increment an accepted event counter (`incEventCnt`). Finally, this class defines a function which is invoked at the end of a run (`endRun`), an abstract function which completes a current exposure (`finishExposure`), which all child classes must implement.

Filter - This class represents a CCD science data filter. It provides functions which track the number of data elements accepted and rejected by a given instance (`accept`, `reject`). It also provides functions that clients use to retrieve and reset this information (`getAcceptCnt`, `getDiscardCnt`, `resetCounters`).

FilterWindow - This class is a subclass of **Filter**, and represents a collection of two dimensional windowing filters. Each window in the collection selects or rejects events or raw pixels based on their position within a CCD. It provides functions to add a window and to remove all windows (`addWindow`, `resetWindows`), to filter a row of raw pixel data (`filterRow`), and to filter a candidate CCD event (`filterEvent`).

FilterGrade - This class is a subclass of **Filter**, and is responsible for filtering events based on their grade code. The grade code of an event represents the spatial pattern of pulse height distributions among the pixels associated with the event. This class provides functions to reject all events (`disableAll`), and to add specific event grades to accept (`allow`). This class also provides a member function which selects or rejects an event based on its grade (`filterEvent`).

FilterPh - This class is a subclass of **Filter**, and is responsible for accepting or rejecting events based on their pulse height. This class provides functions to set the pulse height limits of the filter (`setLimits`), to accept events whose pulse heights are within this range, and to reject those outside the configured limits (`filterEvent`).

FEPexpRec - This is a data structure defined by the Front End to Back End interface. This structure contains exposure information, such as the current exposure number, and the overclock levels being used for the current exposure. See Section 4.10 for a detailed description of the FEP to BEP science interface.

FEPexpEndRec -This is a data structure defined by the Front End to Back End interface. This structure contains information pertaining to the exposure which has just completed,

such as the number of pixels detected above their respective threshold level. See Section 4.10.

FEPerrRec - This is a data structure defined by the FEP to BEP interface. This structure reports the detection of one or two parity errors in the FEP's pixel bias map, and specifies which image pixels are affected. For more detail, see Section 4.10.

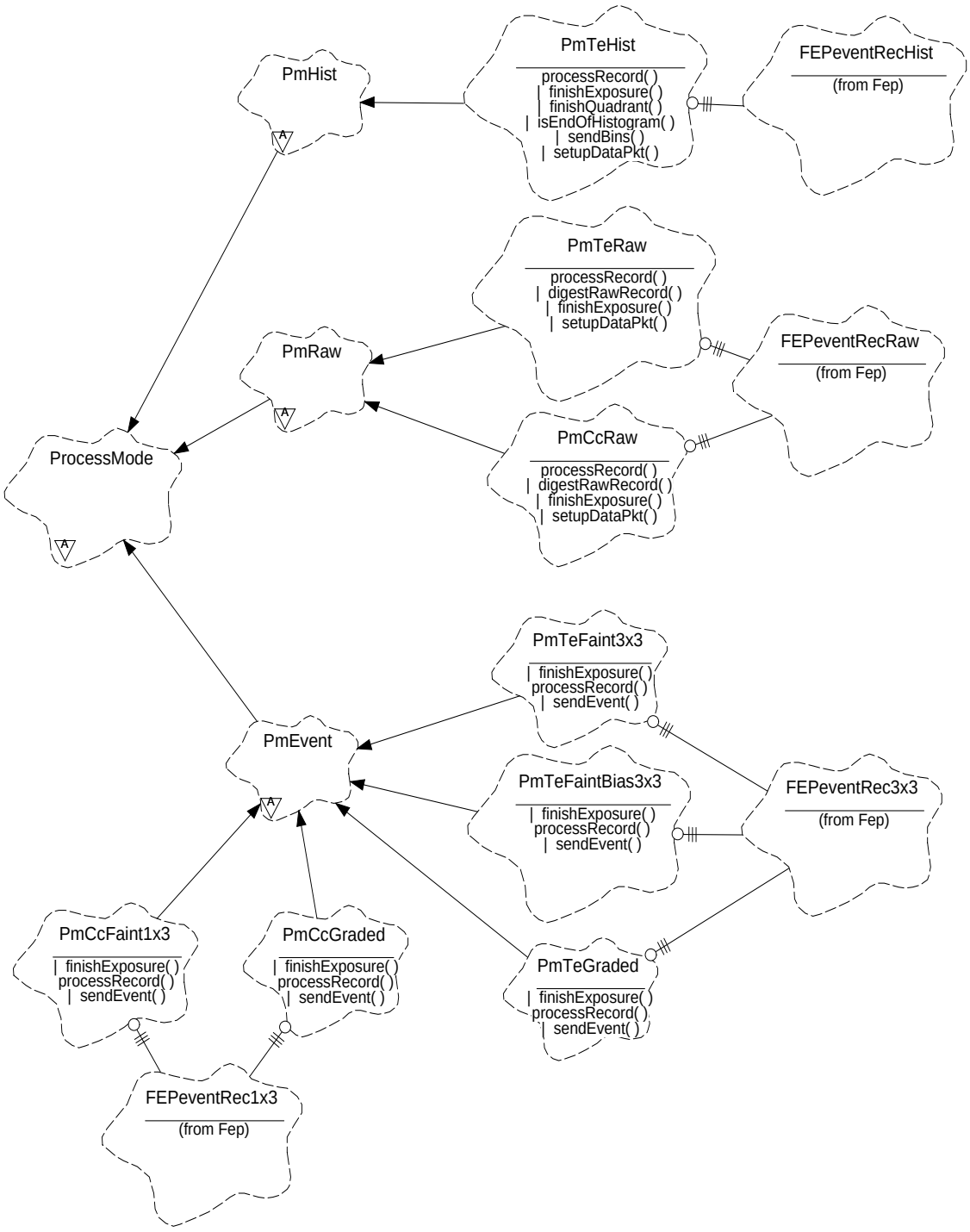
Tf_Data_Bias_Error- This class is responsible for formatting bias error telemetry packets. It is used by the PmEvent class to format and post collections of bias errors.

RINGREC (not shown) - This is an ring buffer data record, used to transport FEP records from the FEP to the BEP via its ring-buffer. This structure consists of an array of thirty-two, 32-bit integers. Usually, this array is cast to the appropriate FEP record type by the consumer of the record.

37.3.2 Process Mode Leaf Classes

Figure 177 illustrates the class relationships between the bottom level data processing classes.

FIGURE 177. Processing Mode Leaf Class Relationships



PmTeHist - This class is a subclass of **PmHist** and is responsible for processing Timed Exposure histogram data produced by a single FEP. In addition to the record types handled by its parent classes, this class processes **FEPeventRecHist** data records, produced by a FEP. This class provides a member function to process FEP to BEP records (processRecord). It also provides member functions used internally to process the end of an exposure (finishExposure), to complete transmission of a histogram from one quadrant (finishQuadrant), to determine if all of the histogram data has been received from the FEP and processed (isEndOfHistogram), to incrementally pack bin data into the telemetry packet buffer (sendBins), and to setup a new telemetry data packet (setupDataPkt).

PmTeRaw - This class is a subclass of **PmRaw** and is responsible for processing Timed Exposure raw pixel data. In addition to the record types handled by its parent classes, this class processes **FEPeventRecRaw** data records produced by a FEP. This class provides a member function to process FEP to BEP records (processRecord). It provides internal member functions to process a complete raw mode data record (digestRawRecord), to setup a new telemetry data packet (setupDataPkt), and to complete an exposure (finishExposure).

PmCcRaw - This class is a subclass of **PmRaw** and is responsible for processing Continuous Clocking raw pixel data. This class handles **FEPeventRecRaw** data records produced by a FEP. This class provides a member function to process FEP to BEP records (processRecord). It provides internal member functions to process a complete raw mode data record (digestRawRecord), to setup a new telemetry data packet (setupDataPkt), and to complete an exposure (finishExposure).

PmTeFaint3x3 - This class is a subclass of **PmEvent** and is responsible for processing **FEPeventRec3x3** event data records produced by a FEP, and producing Timed Exposure Faint Mode exposure records and data packets. This class provides functions which parse data records produced by a FEP (processRecord), accumulate and send processed events using a Timed Exposure Faint Mode data packet (sendEvent), and form and send a Timed Exposure Faint Mode Exposure Record at the end of each exposure (finishExposure).

PmTeFaintBias3x3 - This class is a subclass of **PmEvent**, and is responsible for processing **FEPeventRec3x3** event data records produced by a FEP, and producing Timed Exposure Faint with Bias Mode exposure records and data packets.This class provides functions which parse data records produced by a FEP (processRecord), accumulate and send processed events using a Timed Exposure Faint-with-Bias Mode data packet (sendEvent), and form and send a Timed Exposure Faint-with-Bias Mode Exposure Record at the end of each exposure (finishExposure).

PmTeGraded -This class is a subclass of **PmEvent**, and is responsible for processing **FEPeventRec3x3** event data records produced by a FEP, and producing Timed Exposure Graded Mode exposure records and data packets. This class provides functions which parse data records produced by a FEP (processRecord), accumulate and send processed events using a Timed Exposure Graded Mode data packet (sendEvent), and form

and send a Timed Exposure Faint Mode Exposure Record (used by both Faint and Graded modes) at the end of each exposure (`finishExposure`).

`PmCcFaint1x3` - This class is a subclass of **`PmEvent`**, and is responsible for processing **`FEPeventRec1x3`** event data records produced by a FEP, and producing Continuous Clocking Faint Mode exposure records and data packets. This class provides functions which parse data records produced by a FEP (`processRecord`), accumulate and send processed events using a Continuous Clocking Faint Mode data packet (`sendEvent`), and form and send a Continuous Clocking Faint Mode Exposure Record at the end of each exposure (`finishExposure`).

`PmCcGraded` - This class is a subclass of **`PmEvent`**, and is responsible for processing **`FEPeventRec1x3`** event data records produced by a FEP, and producing Continuous Clocking Graded Mode exposure records and data packets. This class provides functions which parse data records produced by a FEP (`processRecord`), accumulate and send processed events using a Continuous Clocking Graded Mode data packet (`sendEvent`), and form and send a Continuous Clocking Faint Mode Exposure Record (used by both Faint and Graded Modes) at the end of each exposure (`finishExposure`).

`FEPeventRecHist` - This is a data structure defined by the Front End to Back End interface. This structure contains an array of histogram bin values for each output node of the CCD being processed by a FEP. See Section 4.10 for a detailed description of the FEP to BEP science interface.

`FEPeventRecRaw` - This is a data structure defined by the Front End to Back End interface. This structure contains one row of raw pixel values clocked out of a CCD. See Section 4.10 for a detailed description of the FEP to BEP science interface.

`FEPeventRec3x3` - This is a data structure defined by the Front End to Back End interface. This structure contains one candidate 3x3 event detected in the current exposure by the FEP. This record contains the image position of the event, and the 9 pixel pulse heights and bias values corresponding to the event. See Section 4.10 for a detailed description of the FEP to BEP science interface.

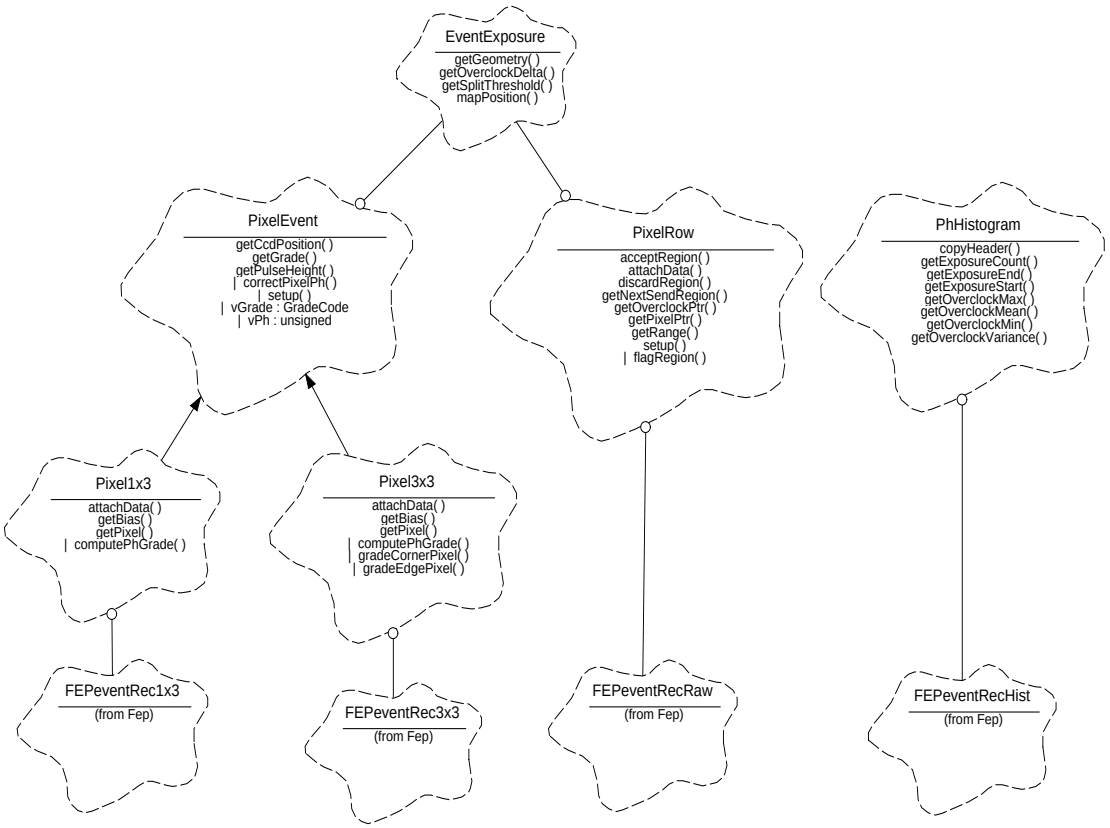
`FEPeventRec1x3` - This is a data structure defined by the Front End to Back End interface. This structure contains one candidate 1x3 event detected in the current exposure by the FEP. This record contains the image position of the event, and the 3 pixel pulse heights and bias values corresponding to the event. See Section 4.10 for a detailed description of the FEP to BEP science interface.

NOTE: 5x5 and 1x5 processing modes are TBD. If implemented the following processing mode classes are needed: **`PmTeFaint5x5`**, **`PmTeFaintBias5x5`**, **`PmCcFaint1x5`**, and **`PmCcFaintBias1x5`**. For robustness, the **`PmTeGraded`** and **`PmCcGraded`** classes would be enhanced to be able to handle **`FEPeventRec5x5`** and **`FEPeventRec1x5`** classes, but their overall functionality would not change.

37.3.3 CCD Data Representation Classes

Figure 178 illustrates the relationships between the data representation classes used for processing CCD events.

FIGURE 178. CCD Data Class Relationships



`PixelEvent` - This class represents a candidate event produced from a CCD and detected by a Front End Processor. Using an **`EventExposure`** instance to supply image geometry information, threshold information, and exposure overclock values, this class provides functions which subclasses use to compute and store the CCD position of the event (`setup`), and the corrected pulse height of a single pixel value (`correctPixelPh`). This class contains two instance variables accessible to its sub-classes which are used to retain the computed pulse height of the event, and the spatial distribution code (grade) of the event (*`vPh`, `vGrade`*). This class provides general client functions to access the position of the event, in CCD coordinates (`getCcdPosition`), the pulse height of the event (`getPulseHeight`), and the spatial distribution grade code of the event (`getGrade`).

`Pixel1x3` - This class is a subclass of **`PixelEvent`** and represents a candidate event detected by a Front End Processor while its CCD is being clocked in Continuous Clocking Mode. This class provides a function to reference pixel values from a **`FEPeventRec1x3`** event record, produced by a FEP (`attachData`). Once the data is loaded, this function computes the pulse height and grade of the event, storing the results in its parent's instance

variables (computePhGrade). This class provides functions which clients use to obtain specific pixel pulse height and bias values (getPixel, getBias).

Pixel3x3 - This class is a subclass of **PixelEvent** and represents a 3x3 pixel event produced by a FEP while a CCD is being clocked in Timed Exposure Mode. It provides a function reference pixel values from a **FEPeventRec3x3** event record produced by a FEP (attachData). Once the data is copied, this function also computes the pulse height of the event and its spatial distribution grade code (computePhGrade, gradeCornerPixel, gradeEdgePixel). This class provides functions which clients use to obtain specific pixel pulse height and bias values (getPixel, getBias).

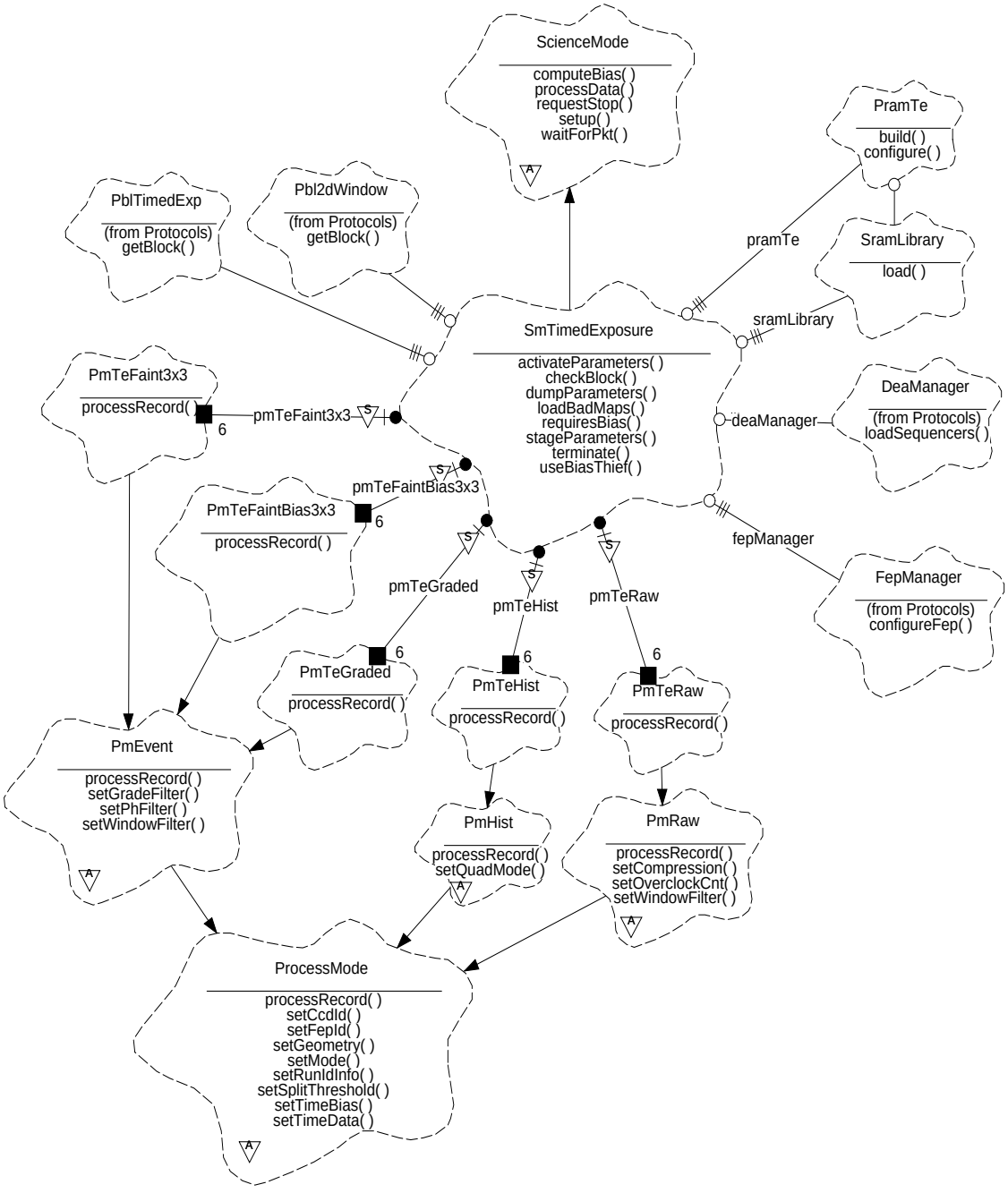
PixelRow - This class represents one row of raw pixel pulse heights. It provides a function to establish a reference to a **FEPeventRecRaw** record produced by a FEP (attachData). Once the reference is established, this function also uses the geometry information in an **EventExposure** instance to map the row position and column range of the row of pixel values. The class provides a function which clients use to obtain the CCD row position and minimum and maximum column positions of the data (getRange). To support windowed clipping of the data, this class provides functions which accept and discard columns within the row (acceptRegion, discardRegion). These functions use an internal function to manipulate an array of flags which indicate which pixels to send (flagRegion). The function provides a function to setup the mask limits based on the number of columns in a row for this image (setup). It provides functions which supplies the next region of a row to be telemetered (getNextSendRegion), and which return a pointer to the pixel buffer array (getPixelPtr) and the overclock pixels (getOverclockPtr).

PhHistogram - This class respresents a histogram of raw pixel pulse heights produced by a FEP. The data is taken from an **FEPeventRecHist** record produced by a FEP. This class provides a function to load the histogram header information (copyHeader). It provides functions to retrieve the number of exposures that went into the histogram (getExposureCount), the first exposure accumulated (getExposureStart), and the last exposure in the histogram (getExposureEnd). It provides functions to obtain a quadrant's overclock maximum and minimum values (getOverclockMax, getOverclockMin), and to obtain a quadrant's overclock mean and variance (getOverclockMean, getOverclockVariance).

37.3.4 Science Mode and Processing Mode Classes

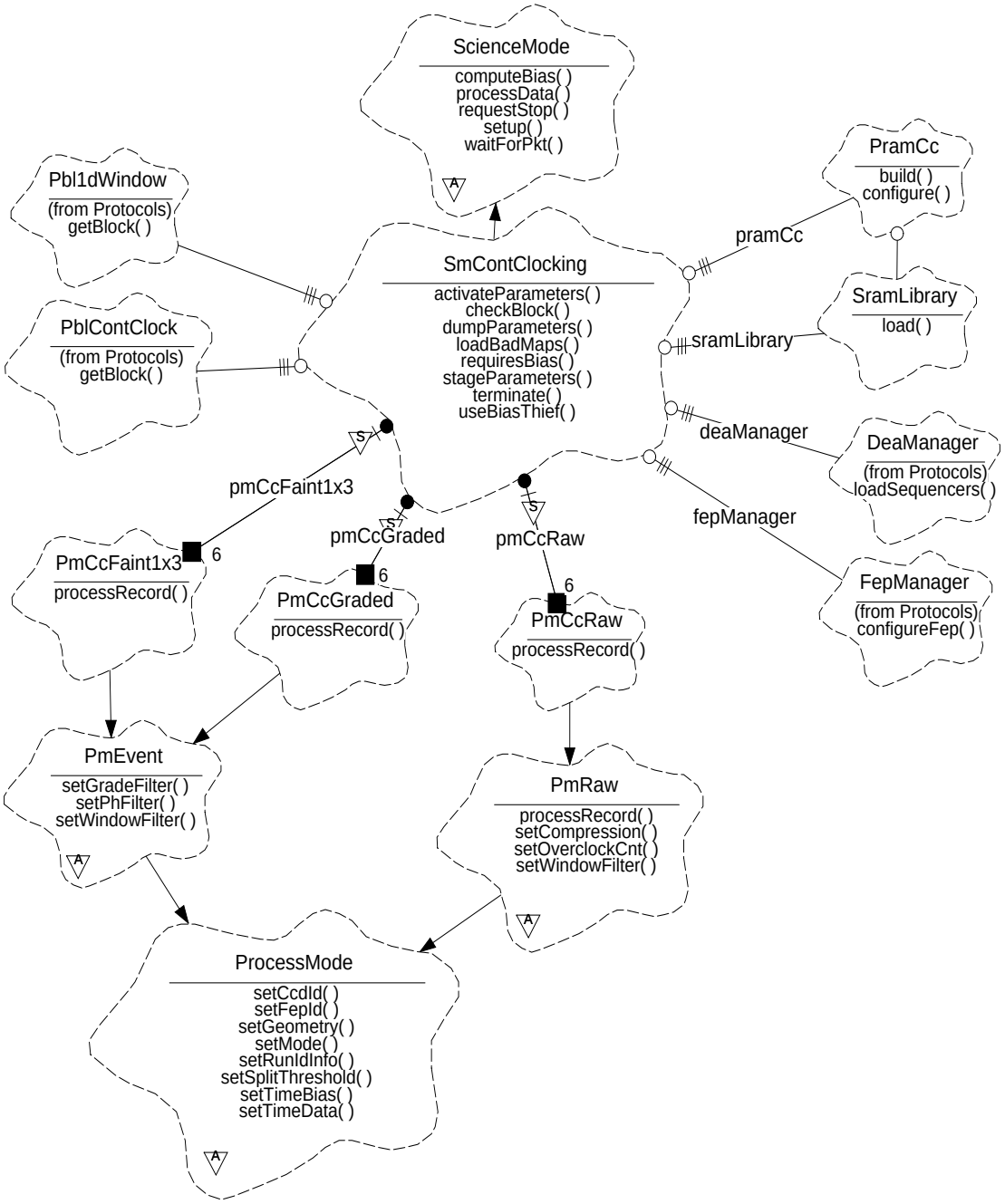
Figure 179 illustrates the relationships between the Timed Exposure Science Mode class and its various data processing modes, and Figure 180 illustrates the relationships between the Continuous Clocking Science Mode and its data processing mode classes.

FIGURE 179. Timed Exposure Mode's Data Processing Class Relationships



SmTimedExposure - This class is a subclass of **ScienceMode** and is responsible for managing a Timed Exposure science run. This class provides functions which stage the mode's parameter blocks prior to starting a run, and activates those parameters once ready to start the run (stageParameters, activateParameters). It provides functions to check the integrity of parameter blocks (checkBlock) and to dump the parameter blocks to telemetry (dumpParameters). This class provides functions which overwrite each FEP's bias map entries with bad pixel entries from the current bad pixel map and timed exposure bad column map (loadBadMaps). It also provides query functions to determine if a bias calibration phase is required for the configured mode, and if so, whether or not the mode requires the bias to be sent to the ground via the **BiasThief** (requiresBias, useBiasThief). This class provides functions used by its parent class to prepare the hardware and software for a particular science run (setupDea, setupFep, setupProcess), and internal functions which it uses to configure the particular details of the run (setupFepBlock, setupRaw, setupHist, setupEvent-Process, setupFaint3x3, setupFaintBias3x3, setupGraded, setupPhFilter, setupWindowFitler, setupGradeFilter). This class uses the **PramTe** and **DeaManager** classes to configure the Detector Electronics Assembly for the run, and the **FepManager** class to identify and configure which Front End Processors to use. When configuring the processing modes, this class selects which **ProcessMode** leaf classes to use (**PmTeRaw**, **PmTeHist**, **PmTeFaint3x3**, **PmTeFaintBias3x3**, **PmTeGraded**), configures the selected instances, and sets up the selected class instances as the FEP process modes for the run. If filters are mandated by the particular processing mode, this class configures the required filter instances, and installs the configured filters into the processing modes. It provides a function which closes the current run and issues a science run report (terminate).

FIGURE 180. Continuous Clocking Mode's Data Processing Class Relationships



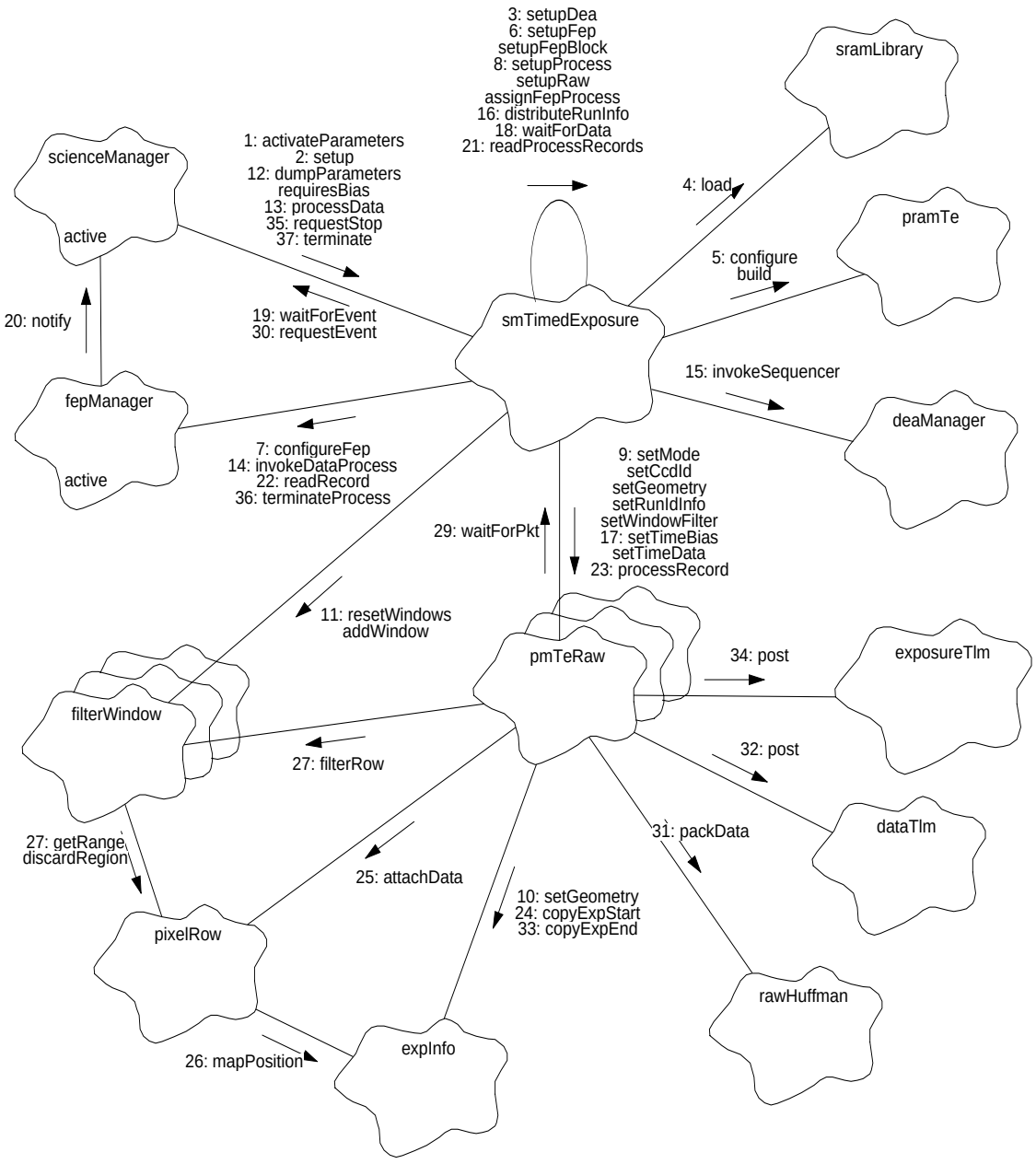
SmContClocking- This class is a subclass of **ScienceMode** and is responsible for managing a Continuous Clocking science run. This class provides functions which stage the mode's parameter blocks prior to starting a run, and activates those parameters once ready to start the run (`stageParameters`, `activateParameters`). It provides functions to check the integrity of parameter blocks (`checkBlock`) and to dump the parameter blocks to telemetry (`dumpParameters`). This class provides functions which overwrite each FEP's bias map entries with bad pixel entries from the current continuous clocking bad column map (`loadBadMaps`). It also provides query functions to determine if a bias is required for the configured mode, and if so, whether or not the mode requires the bias to be sent to the ground via the **BiasThief** (`requiresBias`, `useBiasThief`). This class provides functions used by its parent class to prepare the hardware and software for a particular science run (`setupDea`, `setupFep`, `setupProcess`), and internal functions which it uses to configure the particular details of the run (`setupFepBlock`, `setupRaw`, `setupEventProcess`, `setupFaint1x3`, `setupGraded`, `setupPhFilter`, `setupWindowFitler`, `setupGradeFilter`). This class uses the **PramCc** and **DeaManager** classes to configure the Detector Electronics Assembly for the run, and the **FepManager** class to identify and configure which Front End Processors to use. When configuring the processing modes, this class selects which **ProcessMode** leaf classes to use (**PmCcRaw**, **PmCcFaint1x3**, **PmCcFaintBias1x3**, **PmCcGraded**), configures the selected instances, and sets the selected class instances up as the FEP process modes for the run. If filters are mandated by the particular processing mode, this class configures the required filter instances, and installs the configured filters into the processing modes. It provides a function which closes the current run and issues a science run report (`terminate`).

37.4 Scenarios

37.4.1 Use 1: Timed Exposure Raw Mode Data Processing

Figure 181 illustrates the overall steps involved in configuring and running Timed Exposure Mode, producing raw exposure data.

FIGURE 181. Timed Exposure Raw Mode



1. The *scienceManager* tells Timed Exposure Mode to activate its parameters using *smTimedExposure*.activateParameters(). The parameter blocks have been previously loaded from their parameter block lists via an earlier call to *smTimedExposure*.stageParameters() (not shown), when the initial command to start a run was received. The system of staging and then activating parameters ensures that the correct parameter blocks are used in cases where the start of the run has been deferred due to the radiation monitor, or other reasons.
2. The *scienceManager* tells Timed Exposure Mode to setup for a run, using *smTimedExposure*.setup().
3. setup() calls setupDea() to configure the DEA sequencer logic (PRAM and SRAM).
4. setupDea() loads the SRAM into each DEA CCD-controller, using *sramLibrary*.load().
5. setupDea() generates the sequencer load images by calling *pramTe*.configure() and *pramTe*.build() to build and load the hardware PRAM and SRAM with the images.
6. setup() calls setupFep() to configure the Front End Processors, which calls setupFepBlock() to construct FEP parameter blocks.
7. setupFep() loads the parameter blocks into the FEPs using *fepManager*.configureFep().
8. setup() calls setupProcess() to configure the data process instances. setupProcess() determines from the parameters that raw mode is selected, and calls setupRaw() to configure the filters and process modes. setupRaw() uses assignFepProcess() to install each configured process mode into its table, *fepProcess*[].
9. setupRaw() configures one **PmTeRaw** instance for each Front End Processor being used. It uses setMode() to establish itself as the source of telemetry packet buffers, setCcdId() to identify which CCD a particular instance is dealing with, setGeometry() to establish the clocked image scale and offset factors, setRunIdInfo() to establish the command and parameter block identifiers to report to telemetry, and setWindowFilter() to install the window list filter to use when processing data from the associated CCD.
10. *pmTeRaw*.setGeometry() copies the reported image information into its expInfo instance, using *expInfo*.setGeometry().
11. setupRaw() configures one window filter for each process mode being used. It uses *filterWindow*.resetWindows() to reset the state of a particular filter, and *filterWindow*.addWindow() to configure each clipping window in the filter.
12. Once the mode has been configured, the *scienceManager* instructs the mode to telemeter its parameter blocks, using *smTimedExposure*.dumpParameters(). Once the parameter blocks have been posted to telemetry, the *scienceManager* queries the mode if a bias computation is needed, using

smTimedExposure.requiresBias(). Bias computations are prohibited for raw mode processing, so *smTimedExposure* indicates that no bias computation is needed.

13. The *scienceManager* tells the mode to start clocking, acquiring and processing images using *smTimedExposure*.processData().
14. processData() starts the Front End raw-mode processing functions using *fepManager*.invokeDataProcess().
15. processData() then tells the DEA to start clocking out images, using *deaManager*.invokeSequencer(), which reports the value of the starting micro-second science timestamp.
16. processData() distributes the timestamp to each process mode instance using distributeRunInfo().
17. distributeRunInfo() sets the timestamp of the most recent bias computation using *pmTeRaw*.setTimeBias(), and the current starting timestamp using *pmTeRaw*.setTimeData().
18. processData() then enters its data processing loop, which terminates when the run is stopped. At the top of the loop, processData() calls waitForData() to wait for science data to arrive from one or more of the Front End Processors.
19. waitForData() calls the *scienceManager* to wait for the event using *scienceManager*.waitForEvent().
20. Later, once data becomes available, the *fepManager* notifies the *scienceManager* that data is ready using *scienceManager*.notify(). As a result of the notification, *scienceManager*.waitForEvent() wakes up and returns to *smTimedExposure*.processData()
21. processData() calls readProcessRecords() to read records and process data records from the FEPs.
22. readProcessRecords() calls *fepManager*.readRecord() to read one record from one of the Front End Processors.
23. readProcessRecords() then identifies which FEP produced the data, and feeds a record to the corresponding process mode instance using *pmTeRaw*.processRecord().
24. When processRecord() detects a start of exposure record (**FEPexpRec**), it copies the relevant information using *expInfo*.copyExpStart.
25. When processRecord() detects a raw data record (**FEPeventRecRaw**), it starts constructing a complete raw row record from a series of FEP to BEP records. Subsequent calls to processRecord() append the record data to the partial row record until the record is complete. Once the record is complete, processRecord() processes the completed record using digestRawRecord() (not shown). It establishes a **PixelRow** instance, and associates the record using *pixelRow*.attachData().
26. attachData() uses the *expInfo* structure to map the image row coordinates into absolute CCD coordinates using *expInfo*.mapPosition().

27. *pmTeRaw* then filters the row using *filterWindow*.filterRow().

28. filterRow() uses *pixelRow*.getRange() to find the position limits of the raw pixels, and, if needed, discards regions of the row using *pixelRow*.discardRegion().

29. *pmTeRaw*.digestRawRecord() checks its data telemetry object, *dataTlm*, to see if it has a telemetry buffer (not shown). If not, *pmTeRaw* asks its science mode to wait for and allocate a telemetry packet buffer, by passing *dataTlm* to *smTimedExposure*.waitForPkt().

30. waitForPkt() tells the passed telemetry object to wait for and allocate a packet. Whenever the time-out expires, waitForPkt() checks the science manager for certain pending events, such as task monitor queries, using *scienceManager*.requestEvent(), responds to the requests, and retries to obtain a telemetry packet buffer.

31. Once *dataTlm* has a buffer, *pmTeRaw*.digestRawRecord() sets the header information in the packet, and compresses the data directly into the packet buffer's data area using *rawHuffman*.packData().

32. Once the *dataTlm*'s buffer is full, or the maximum number of rows have been packed into the buffer, digestRawRecord() posts the telemetry objects's buffer to telemetry using *dataTlm*.post().

33. When processRecord() detects the end of an exposure (**FEPexpEndRec**), it copies the relevant information using *expInfo*.copyExpEnd(), and invokes finishExposure() (not shown).

34. finishExposure() then establishes an exposure record telemetry object, *exposureTlm*, and obtains a telemetry packet buffer for the record. It then fills the exposure record header and tells the object to post its telemetry buffer for transfer, using *exposureTlm*.post().

35. Later, when a command is received to stop the run, the *scienceManager*'s binding function tells the mode to stop the current run, using *smTimedExposure*.requestStop(). Then, requestStop() sets an internal flag and notifies the task that a stop has been requested.

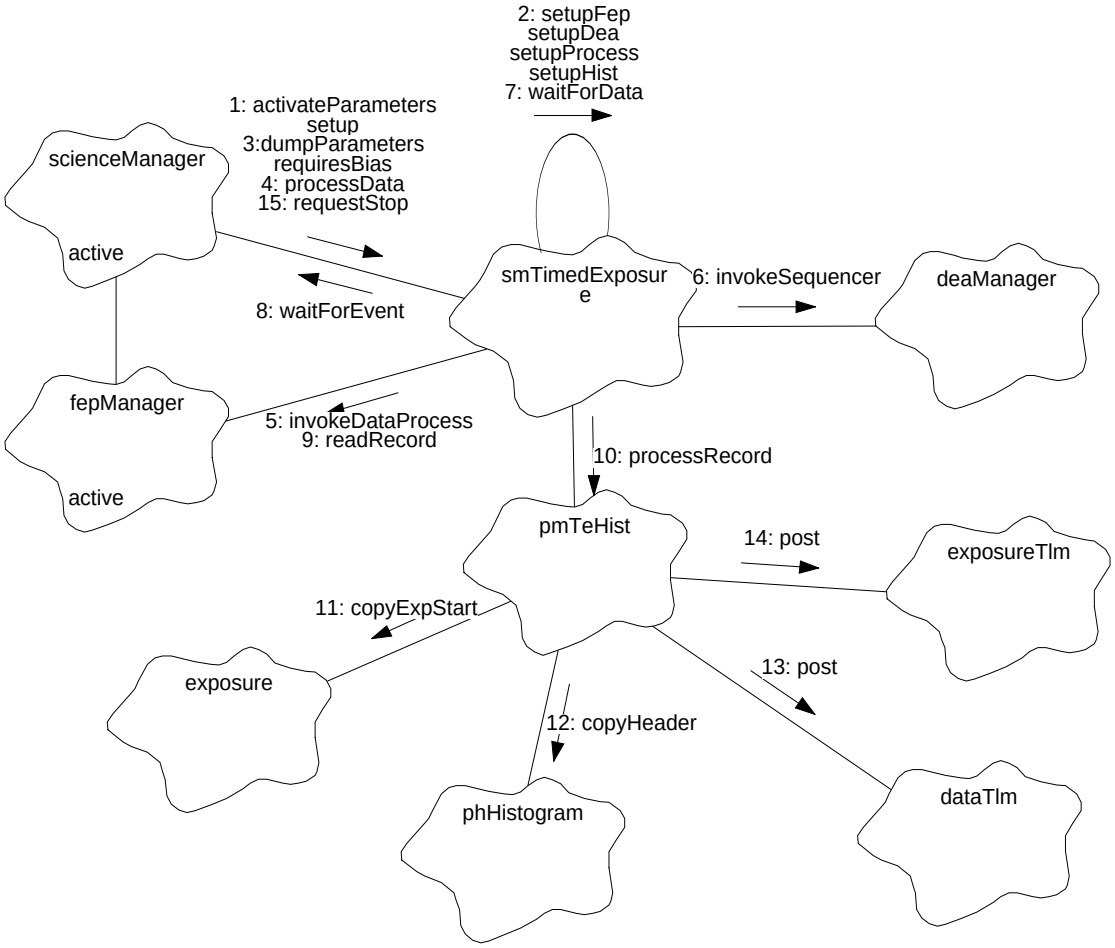
36. If *smTimedExpousre*.processData() has called waitForData() to wait for data to arrive, the wait call aborts to allow processData to respond to the stop request. *smTimedExposure*.processData()'s loop detects the stop request in the internal flag, and tells the FEPs to finish up processing, using *fepManager*.terminateProcess(). It then continues its processing loop, waiting for and handling FEP data records, until the FEPs report that they are done (not shown). Once the FEPs have finished producing data, and all of the data records have been consumed, processData() stops the DEA sequencers using *deaManager*.stopSequencer() (not shown).

37. Once processData() returns, *scienceManager* instructs the mode to cleanup from the run, using *smTimedExposure*.terminate().

37.4.2 Use 2: Timed Exposure Histogram Mode Data Processing

Figure 182 illustrates the overall steps involved in configuring and running Timed Exposure Mode, producing histograms of raw pixel pulse heights. Some of the setup and processing details are the same as those described for Raw Mode processing (see Section 37.4.1) and are omitted from the diagram and description.

FIGURE 182. Timed Exposure Raw Histogram Mode



1. The *scienceManager* object sets up for a Timed Exposure Science Run, calling *smTimedExposure*.activateParameters() and *smTimedExposure*.setup().
2. *smTimedExposure*.setup() calls setupDea(), setupFep, and setupProcess() to configure the hardware and the internal structures for the run. setupProcess() determines that Histogram processing was selected and calls setupHist(), which selects the set of **PmTeHist** instances and configures each instance.

3. *scienceManager* initiates a parameter dump, using *smTimedExposure*.dumpParameters(). *scienceManager* determines if a bias computation is needed by calling *smTimedExposure*.requiresBias().
4. *smTimedExposure*.processData() starts data processing on each of the configured FEPs using *fepManager*.invokeDataProcess().
5. It then starts the sequencers using *deaManager*.invokeSequencer(), and stores the science microsecond timestamp as the start time of the data processing run.
6. processData() then waits for data to arrive from the FEPs using waitForData().
7. waitForData() calls the *scienceManager* to wait for the event using *scienceManager*.waitForEvent().
8. Once data arrives, processData() reads one record from one of the FEPs using *fepManager*.readRecord().
9. processData() then passes the record to *pmTeHist*.processRecord() to be processed.
10. Whenever processRecord() reads an Exposure Start Record (**FEPexpRec**), it passes it to *expInfo*.copyExpStart() to extract exposure information, such as the overclock values used for the reported exposure.
11. When processRecord() reads a Histogram Event Data Record (**FEPeventRecHist**), it copies the header information from the record using *phHistogram*.copyHeader(). Then proceeds to accumulate and store bin data as it arrives from subsequent records from the FEP.
12. For each data packet it creates, *pmTeHist* stores the starting bin number into the telemetry packet buffer. It the adds the bin values directly into the telemetry buffer. Whenever a buffer becomes full, or when the histogram for one of the quadrants has been completed, *pmTeHist* posts the data packet buffer to telemetry using *dataTlm*.post().
13. Once the histogram from a quadrant has been posted, *pmTeHist* forms a histogram record telemetry packet buffer, fills in the packet information and posts it to telemetry using *exposureTlm*.post().
14. Eventually, the *scienceManager* instructs the mode to stop the run, using *smTimedExposure*.requestStop(). Once the FEPs have completed the last histogram, and their data have been posted for transfer, the *scienceManager* tells the mode to cleanup from the run, using *smTimedExposure*.terminate() (not shown).

37.4.3 Use 3: Timed Exposure Faint Mode 3x3 Event Data Processing

Figure 183 illustrates the overall steps involved in configuring and running Timed Exposure Mode, producing Faint Mode 3x3 event lists. Some of the setup and processing details are the same as those described for Raw Mode processing (see Section 37.4.1) and are omitted from the diagram and description. All of the other event processing modes have the same overall structure, and their descriptions refer to this mode.

FIGURE 183. Timed Exposure Faint 3x3 Event Mode

1. The *scienceManager* object sets up for a Timed Exposure Science Run, calling *smTimedExposure*.activateParameters() and *smTimedExposure*.setup().

2. *smTimedExposure*.setup() calls setupDea(), setupFep, and setupProcess() to configure the hardware and the internal structures for the run.
3. setupProcess() determines that Faint Mode 3x3 Event processing was selected and calls setupFaint3x3(), which selects the set of **PmTeFaint3x3** instances, and calls setupEventProcess() to configure each instance. setupEventProcess() configures a set of pulse height, window and grade filters using setupPhFilter(), setupWindowFilter(), and setupGradeFilter() respectively.
4. setupEventProcess() configures each *pmTeFaint3x3* object, installing the instance's pulse height, window and grade filters using setPhFilter(), setWindowFilter(), and setGradeFilter() respectively. setupEventProcess() then installs the process mode into its array of pointers (not shown).
5. *scienceManager* initiates a parameter dump, using *smTimedExposure*.dumpParameters(). *scienceManager* determines if a bias computation is needed by calling *smTimedExposure*.requiresBias(). Assume for this example that the parameter block mandates a bias computation. Upon determining that a bias computation is needed, the *scienceManager* starts the bias computations using *smTimedExposure*.computeBias().
6. *smTimedExposure*.computeBias() starts the bias computation process on each of the configured Front Ends, using *fepManager*.invokeBiasProcess().
7. Once the FEPs are ready to receive images, computeBias() starts the DEA sequencers, using *deaManager*.invokeSequencer(), storing the returned science microsecond timestamp as the start time of the most recent bias computation (a static instance variable of **ScienceMode** maintained in D-cache).
8. computeBias() then waits for the bias computations to complete on all of the FEPs, using waitForBias().
9. waitForBias() uses the *scienceManager*.waitForEvent() to block execution until the bias computations complete. Later, once all of the active FEPs have completed their bias computations, the *fepManager* notifies the *scienceManager* (not shown). After the bias computations have completed, computeBias() shuts down the sequencers using *deaManager*.stopSequencer() (not shown).
10. The *scienceManager* then overwrites the pixel bias values corresponding to bad pixels and columns using *smTimedExposure*.loadBadMaps(). Once the maps are loaded, the *scienceManager* starts data processing, calling *smTimedExposure*.processData().
11. *smTimedExposure*.processData() starts data processing on each of the configured FEPs using *fepManager*.invokeDataProcess(). It then starts the sequencers using *deaManager*.invokeSequencer() (not shown), and stores the science microsecond timestamp as the start time of the data processing run.
12. processData() then sets the start times within each of the process modes, using *pmTeFaint3x3*.setTimeBias() and *pmTeFaint3x3*.setTimeData().
13. processData() then waits for data to arrive from the FEPs using waitForData().

14. Once data arrives, `processData`() reads one or more records from one of the FEPs using *`fepManager`*.`readRecord`(). and calls *`pmTeFaint3x3`*.`processRecord`() to process the acquired FEP record.

15. Whenever `processRecord`() reads an Exposure Start Record (**`FEPexpRec`**), it passes it to *`expInfo`*.`copyExpStart`() to extract exposure information, such as the overclock values used for the reported exposure.

16. When `processRecord`() reads a 3x3 Event Data Record (**`FEPeventRec3x3`**), it forms a temporary **`Pixel3x3`** instance, and loads it with the data record, using *`pixel3x3`*.`attachData`().

17. `attachData`() maps the image row and column position into absolute CCD coordinates, using *`expInfo`*.`mapPosition`(). It then, for each of the three pixel columns of the event, loads the exposure overclock information, using *`expInfo`*.`getOverclockDelta`(), and loads the configured split threshold levels using *`expInfo`*.`getSplitThresholds`().

18. `attachData`() then computes the event's pulse height and grade using `computePhGrade`(). For each pixel, `computePhGrade`() uses `correctPixelPh`() (not shown) to determine the corrected pulse height of a pixel, based on the raw pixel pulse height, pixel bias level and overclock level. `computePhGrade`() then uses `gradeEdgePixel`() (not shown) to compare the corrected value against the split threshold. If the value is greater than or equal to its column's split threshold, it sets the pixel's grade bit in the grade code and adds its pulse height to the current sum for the event. It then sets flags indicating that the adjacent corner pixels are permitted to contribute to the pulse height of the event. Once the edge pixels have been evaluated, `computePhGrade`() uses `gradeCornerPixel`() (not shown) to process each corner pixel of the 3x3 event. `gradeCornerPixel`() compares the pixel's corrected pulse height to its split threshold. If it is greater than or equal to the split threshold, it sets the pixel's grade bit in the grade code. It then determines if the corner pixel is adjacent to an edge pixel above split threshold. If so, it adds the corner pixel's corrected pulse height to the total energy of the event. If not, although the pixel's grade bit may be set, the corner pixel's pulse height is not added to the total energy.

19. *`pmTeFaint3x3`*.`processRecord`() then calls *`pmTeFaint3x3`*.`filterEvent`() (not shown) to run the event through its filter set. `filterEvent`() determines if the event should be telemetered based on its pulse height using *`filterPh`*.`filterEvent`().

20. *`filterPh`*.`filterEvent`() queries the computed pulse height of the event using *`pixel3x3`*.`getPulseHeight`(), and compares the value with its configured limits. If the pulse height is within its limits, the event is accepted. If not, the event is rejected.

21. If the pulse height filter accepts the event, *`pmTeFaint3x3`*.`filterEvent`() calls *`filterWindow`*.`filterEvent`() to test the event against the configured 2-D windows.

22. *filterWindow*.filterEvent() uses *pixel3x3*.getCcdPosition() to obtain the CCD row and column of the center of the event. It then checks the position against each of its configured windows. If the event is within the bounds of one of its windows, it checks the sample counter for the window against the window's limit, and increments the counter. If the limit is 0, or if the counter is at the configured limit, *filterWindow*.filterEvent() gets the pulse height of the event using *pixel3x3*.getPulseHeight() and resets the counter. If the pulse height is within the bounds of the event, the event is accepted for further processing. If either the sample counter has not reached its limit, or if the pulse height of the event is outside the bounds configured for the window, the event is rejected.

23. If the window filter accepts the event, *pmTeFaint3x3*.filterEvent() calls *filterGrade*.filterEvent() to test the event's grade against the set of accepted grades.

24. *filterGrade*.filterEvent() calls *pixel3x3*.getGrade() to obtain the computed grade code of the event. If the code is in the filters list of accepted grades, the event is accepted, otherwise it is rejected.

25. If the event is accepted by all of the event filters, processRecord() calls sendEvent() (not shown) to add the event to its data packet buffer. sendEvent() checks the state of its data telemetry object, *dataTlm*. If it does not have a telemetry packet buffer, sendEvent() obtains the packet buffer (see Section 37.22.4), and sets the packet's data header information. It then tells the packet to pack the event into its data area, using *dataTlm*.append_Event().

26. sendEvent() determines if the packet's buffer becomes full using *dataTlm.*isFull(). If the packet becomes full, it posts the packet's data to telemetry using *dataTlm*.post().

27. When processRecord() receives an End of Exposure record (**FEPexpEndRec**), it uses *expInfo*.copyExpEnd() (not shown) to extract the ending exposure information from the record. It then establishes an exposure record telemetry object, waiting for and allocating a telemetry packet buffer (not shown). Once the object has a buffer, it stores the exposure header information into the packet buffer. Once the exposure record is complete, its telemetry packet buffer is posted for transfer to telemetry using *exposureTlm*.post().

28. Eventually, the *scienceManager* instructs the mode to stop the run, using *smTimedExposure*.requestStop(). Once the FEPs have completed the last exposure, and their data have been posted for transfer, the *scienceManager* tells the mode to cleanup from the run, using *smTimedExposure*.terminate().

37.4.4 Use 4: Timed Exposure Faint Mode 3x3 with Bias Data Processing

The overall operation of this data processing mode is identical to that used by Faint 3x3 Mode (see Section 37.4.3), except that a **PmTeFaintBias3x3** instance is used, the telemetry packet formats are slightly different, and the sendEvent() member function of **PmTeFaintBias3x3** adds event bias information to the data telemetry packet, which is not present in Faint Mode.

37.4.5 Use 5: Timed Exposure Graded Mode Data Processing

The overall operation of this data processing mode is identical to that used by Faint 3x3 Mode (see Section 37.4.3), except that a **PmTeGraded** instance is used, the telemetry packet formats are slightly different, and the sendEvent() member function of **PmTeGraded** sends event amplitude, grade, and corner pulse height sums instead of the raw pixel pulse heights sent by Faint Mode.

37.4.6 Use 6: Continuous Clocking Raw Mode Data Processing

The overall operation of Continuous Clocking Raw Mode is identical to that used by Timed Exposure Raw Mode (see Section 37.4.1), except that *pramCc* is used to configure the sequencers PRAM, a **PmCcRaw** instance is used to process the raw data, and the low-level details of the setup functions are slightly different, given the different nature of the parameter blocks, CCD data, and telemetry formats.

37.4.7 Use 7: Continuous Clocking Faint Mode 1x3 Event Data Processing

The overall operation of this data processing mode is identical to that used by Timed Exposure Faint 3x3 Mode (see Section 37.4.3), except the mode used is **SmContClocking**, a **PmCcFaint1x3** instance is used, a **Pixel1x3** class is used to represent the data instead of a **Pixel3x3** class, the telemetry packet formats are slightly different, and the sendEvent() member function of **PmCcFaint1x3** produces 1x3 pixel pulse height data instead of 3x3 event data.

37.4.8 Use 8: Continuous Clocking Graded Mode Data Processing

The overall operation of this data processing mode is identical to that used by Timed Exposure Faint 3x3 Mode (see Section 37.4.3), except the mode used is **SmContClocking**, a **PmCcGraded** instance is used, a **Pixel1x3** class is used to represent the data instead of a **Pixel3x3** class, the telemetry packet formats are slightly different, and the sendEvent() member function of **PmCcGraded** produces events represented by a event amplitude, and a 2-bit grade code.

.

37.5 Class SmTimedExposure

Documentation:

This class represents the Timed Exposure Science Mode and is responsible for providing member functions which configure and run a Timed Exposure science and/or bias run.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **ScienceMode**

Implementation Uses:

PblTimedExp
DeaManager
FepManager
Pbl2dWindow
PramTe
SramLibrary
Tf_Dump_Te_Block

Public Interface:

Operations:

```
SmTimedExposure()
activateParameters()
checkBlock()
dumpParameters()
loadBadMaps()
requiresBias()
stageParameters()
terminate()
useBiasThief()
```

Protected Interface:

Has-A Relationships:

PmTeFaintBias3x3 *pmTeFaintBias3x3*[6]: These are the process modes used to process Timed Exposure 3x3 Faint-with-Bias Mode event data, indexed by FEP Id.

PmTeGraded *pmTeGraded*[6]: These are the process modes used to process Timed Exposure Graded Mode event data, indexed by FEP Id.

PmTeHist *pmTeHist*[6]: These are the process modes used to process Timed Exposure histograms, indexed by FEP Id.

PmTeRaw *pmTeRaw*[6]: These are the process modes used to process Timed Exposure Raw mode data, indexed by FEP Id.

PmTeFaint3x3 *pmTeFaint3x3*[6]: These are the process modes used to process Timed Exposure 3x3 Faint Mode event data, indexed by FEP Id.

Operations:

```
getBlockIds()
getFepRequest()
setupDea()
setupEventProcess()
setupProcess()
setupFaint3x3()
setupFaintBias3x3()
setupFep()
setupFepBlock()
setupGraded()
setupGradeFilter()
setupHist()
setupPhFilter()
setupRaw()
setupWindowFilter()
```

Private Interface:

Has-A Relationships:

CmdPkt_Load_Te_Block *teBlock*: This is the active Timed Exposure Parameter Block being used for the current run.

CmdPkt_Load_Te_Block *stagedTeBlock*: This is the next Timed Exposure Parameter block to use. This block is setup by the command requesting the start of a new run, and is copied into teBlock once the run starts.

CmdPkt_Load_2D_Block *winBlock*: This is the active 2D Window Parameter Block, if one is being used.

CmdPkt_Load_2D_Block *stagedWinBlock*: This is the window parameter block to use for the next run. It is setup by the handler which requested the new run, and is copied into winBlock once the run starts.

Boolean *hasWindows*: This flag indicates whether or not the active parameter block is using window filters.

CcdId *fepCcd*: This is an array of CCD identifiers extracted from the active parameter block. The array is indexed by FEP Id.

Boolean *fepVideo*[6]: This is an array of FEP video gain selection, indexed by FEP Id and extracted from the active parameter block. BoolFalse indicates that the FEP's CCD should be clocked using 1electron/ADU. BoolTrue indicates that the CCD should be clocked to obtain 4 electrons/ADU.

int *fepThresh*[6][4]: This is an array of FEP quadrant threshold settings. The settings are indexed by FEP Id and video chain.

unsigned*fepSplit*[6][4]: This is an array of split threshold settings, indexed by FEP. These are copied from the active parameter block.

unsigned *ccdsUsed*: This specifies the total number of CCDs being used for the active run.

unsigned *ccdSlot*[10]: This is an array, indexed by CCD Id, which maps a CCD to a particular Image to Frame transfer slot for the active run.

Boolean *biasOk*: This indicates if the bias maps are intact. If *BoolTrue*, then all configured bias maps are ready. If *BoolFalse*, then one or more bias maps need to be built.

Concurrency: Synchronous

Persistence: Persistent

37.5.1 SmTimedExposure()

Public member of: **SmTimedExposure**

Documentation:

This is the constructor for the **SmTimedExposure** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.5.2 activateParameters()

Public member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function copies the parameter blocks staged in *stagedTeBlock* and *stagedWinBlock* into *teBlock* and *winBlock*.

Semantics:

Get the buffer address of the staged and main blocks, and get the length of the staged block and copy the data from the staged block into the main block. Then check the block if a window is specified, and if so, copy the staged window block into the active window block. Using the now active block, cache the FEP to CCD selections, FEP video selections and FEP threshold and split threshold selections into *fepCcd*, *fepVideo*, *fepThresh*, and *fepSplit*, respectively. Then iterate through the FEP CCD selections and verify that the corresponding DEA and FEP boards have power. If a DEA board or FEP board isn't powered, invalidate the corresponding FEP's CCD selection to prevent the run from using the FEP. If the boards are on, store the current active CCD count as the clocking parallel transfer slot to use when clocking the chip, and advance the total number of chips configured for use during the run.

Concurrency: Guarded

37.5.3 checkBlock()

Public member of: **SmTimedExposure**

Return Class: **Boolean**

Arguments:

unsigned *blockid*

Documentation:

This function checks the integrity of the Timed Exposure parameter block, indicated by *blockid*. If the parameter block is intact, the function returns *BoolTrue*. If the block has been corrupted, the function returns *BoolFalse*.

Concurrency: Synchronous

37.5.4 dumpParameters()

Public member of: **SmTimedExposure**

Return Class: **Boolean**

Documentation:

This function posts the contents of the active Timed Exposure and 2D window parameters to telemetry. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

Declare a Timed Exposure Parameter block form, and use waitForPkt() to obtain a telemetry packet buffer for the form. If the run is aborted while waiting for the buffer, return *BoolFalse*. Once a buffer has been obtained, copy the active Timed Exposure Parameter Block into the buffer. If a window list is used, append the active 2D Window list to the buffer. Then post the buffer for transfer to telemetry.

Concurrency: Guarded

37.5.5 getBlockIds()

Protected member of: **SmTimedExposure**

Return Class: **void**

Arguments:

unsigned& *blockId*
unsigned& *winId*

Documentation:

This function retrieves the parameter block ids from the Timed Exposure Parameter block and the 2D Window parameter block (if used). On return, *blockId* will contain the parameter block id from the active Timed Exposure Parameter Block. If a 2D window is used, *winId* will contain its parameter block id. If not, *winId* will contain 0xffffffff.

Concurrency: Guarded

37.5.6 getFepRequest()

Protected member of: **SmTimedExposure**

Return Class: **int**

Documentation:

This function returns the FEP request code used to start Timed Exposure science data processing.

Concurrency: Guarded

37.5.7 loadBadMaps()

Public member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function loads the bad pixels and timed exposure bad columns into their respective FEP bias maps.

Concurrency: Guarded

37.5.8 requiresBias()

Public member of: **SmTimedExposure**

Return Class: **Boolean**

Documentation:

This function determines if a bias computation is required. It first checks the parameter block for a bias request. If a bias is not mandated by the parameter block, it then queries each configured FEP to ensure that each has a valid bias. If either the parameter block mandates a bias computation, or if a FEP has an invalid bias, the function returns *BoolTrue*. If no bias is required by the configured mode, then it returns *BoolFalse*.

Concurrency: Guarded

37.5.9 setupDea()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function loads the Timed Exposure SRAM image into the DEA CCD-controllers, and uses the PRAM Builder to build and load the Timed Exposure clocking operations into the CCD-controller's PRAM.

Semantics:

If the run has a custom DEA load block, use *deaManager*.loadSequencers() to load it into the CCD controllers. If not, iterate through each configured FEP. Load the SRAM library using *sramLibrary*.load(), configure the PRAM builder and build and load the PRAM using *pramTe*.configure() and *pramTe*.build().

Concurrency: Guarded

37.5.10 setupEventProcess()

Protected member of: **SmTimedExposure**

Return Class: **void**

Arguments:

FepId *fep*
PmEvent& *process*

Documentation:

This function configures the event process, indicated by *process*, to handle events produced by the FEP, *fep*. It configures the run-header information, CCD geometry, sets up the filters, installs the filters, and installs *process* in the mode's CCD processor pointer array.

Concurrency: Guarded

37.5.11 setupFaint3x3()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up for Faint 3x3 event processing. This function iterates through each configured FEP, passing a separate **PmTeFaint3x3** instance to setupEventProcess() for each FEP being used.

Concurrency: Guarded

37.5.12 setupFaintBias3x3()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up for Faint 3x3 with Bias event processing. This function iterates through each configured FEP, passing a separate **PmTeFaintBias3x3** instance to setupEventProcess() for each FEP being used.

Concurrency: Guarded

37.5.13 setupFep()

Protected member of: **`SmTimedExposure`**

Return Class: **`void`**

Documentation:

This function loads code into each of the configured FEPs, and sets up the Timed Exposure mode parameters into each.

Semantics:

For each configured FEP, first load and run any customized code into the FEP, using *`fepManager`*`.loadRunProgram`(). Then query the status of the FEP, using *`fepManager`*`.queryFepStatus`(). If the query failed, then the FEP has crashed. Restart the FEP using the default program load. Use `setupFepBlock`() to build the BEP to FEP parameter block, and use *`fepManager`*`.configureFep`() to load the configuration into the FEP.

Concurrency: Guarded

37.5.14 setupFepBlock()

Protected member of: **SmTimedExposure**

Return Class: **Boolean**

Arguments:

FepId *fep*
FEPparmBlock& *fepblock*

Documentation:

This function sets up a Front End Processor Parameter Block, *fepblock*, for the FEP specified by *fep*. If successful, the function returns *BoolTrue*. If the FEP did not respond, it returns *BoolFalse*.

Concurrency: Guarded

37.5.15 setupGradeFilter()

Protected member of: **SmTimedExposure**

Return Class: **void**

Arguments:

FepId *fep*
FilterGrade& *grade*

Documentation:

This function sets up the Grade filter, *grade*, using the active grade selection. The passed FEP identifier, *fep*, is a hook to support enhancements which make the grade selection FEP-specific.

Concurrency: Guarded

37.5.16 setupGraded()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up for Graded event processing.This function iterates through each configured FEP, passing a separate **PmTeGraded** instance to `setupEventProcess`() for each FEP being used.

Concurrency: Guarded

37.5.17 setupHist()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up for Histogram data processing.This function iterates through each configured FEP. For each FEP in use, it sets the mode, the CCD Id, and the geometry into a separate **PmTeHist**, and passes the instance to `assignFepProcess`() to associate the instance with the FEP.

Concurrency: Guarded

37.5.18 setupPhFilter()

Protected member of: **SmTimedExposure**

Return Class: **void**

Arguments:

FepId *fep*
FilterPh& *phFilter*

Documentation:

This function configures the Pulse Height filter, *phFilter*. The passed FEP identifier, *fep*, is a hook to support changes which would make the pulse height selections FEP-specific.

Concurrency: Guarded

37.5.19 setupProcess()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up the processing objects used to process data for the run.

Semantics:

This function examines the configured FEP and BEP processing mode selections. If performing `raw` mode, call `setupRaw`(). If performing histogram mode, call `setupHist`(). If performing 3x3 Faint Mode, call `setupFaint3x3`(). If performing Faint-with-Bias, call `setupFaintBias3x3`(), and if performing Graded Mode, call `setupGraded`().

Concurrency: Guarded

37.5.20 setupRaw()

Protected member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function sets up for Raw data processing. For each FEP in use, it sets the mode, the CCD Id, geometry, into a separate **PmTeRaw** instance. It also sets up a distinct window list filter and assigns it to the **PmTeRaw** instance. It then passes the instance to assignFepProcess() to associate the instance with the FEP.

Concurrency: Guarded

37.5.21 setupWindowFilter()

Protected member of: **SmTimedExposure**

Return Class: **void**

Arguments:

FepId *fep*
FilterWindow& *window*

Documentation:

This function sets up the window list filter, *window*, using the active window parameters used by the CCD being processed by the FEP, *fep*.

Concurrency: Guarded

37.5.22 stageParameters()

Public member of: **SmTimedExposure**

Return Class: **void**

Arguments:

unsigned *blockid*

Documentation:

This function fetches the Timed Exposure parameter block, indicated by *blockid*, and stores its contents into *stagedTeBlock*. If a window block is referenced within the block, it is fetched and stored into *stagedWinBlock*.

Concurrency: Synchronous

37.5.23 terminate()

Public member of: **SmTimedExposure**

Return Class: **void**

Documentation:

This function cleans up after a Timed Exposure science run. It produces and posts a science run report.

Concurrency: Guarded

37.5.24 useBiasThief()

Public member of: **SmTimedExposure**

Return Class: **Boolean**

Documentation:

This function determines whether or not the Bias Thief should be invoked. It returns *BoolTrue* if the bias thief should send the pixel bias values, and *BoolFalse* if not.

Concurrency: Guarded

37.6 Class SmContClocking

Documentation:

This class represents the Continuous Clocking Science Mode and is responsible for providing member functions which configure and run a Continuous Clocking science and/or bias run.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **ScienceMode**

Implementation Uses:

DeaManager
FepManager
Pbl1dWindow
PramCc
SramLibrary
PblContClock
Tf_Dump_Cc_Block

Public Interface:

Operations:

```
SmContClocking()
activateParameters()
checkBlock()
dumpParameters()
loadBadMaps()
requiresBias()
stageParameters()
terminate()
useBiasThief()
```

Protected Interface:

Has-A Relationships:

PmCcGraded *pmCcGraded*[6]: These are the process modes used to process Continuous Clocking Graded Mode event data, indexed by FEP Id.

PmCcRaw *pmCcRaw*[6]: These are the process modes used to process Continuous Clocking Raw Mode event data, indexed by FEP Id.

PmCcFaint1x3 *pmCcFaint1x3*[6]: These are the process modes used to process Continuous Clocking Faint Mode event data, indexed by FEP Id.

Operations:

```
getBlockIds()
getFepRequest()
setupDea()
setupEventProcess()
setupFaint1x3()
setupFep()
setupFepBlock()
setupGradeFilter()
setupGraded()
setupPhFilter()
setupProcess()
setupRaw()
setupWindowFilter()
```

Private Interface:

Has-A Relationships:

`CmdPkt_Load_Cc_Block` *`ccBlock`*: This is the active Continuous Clocking Parameter Block being used for the current run.

`CmdPkt_Load_Cc_Block` *`stagedCcBlock`*: This is the next Continuous Clocking Parameter block to use. This block is setup by the command requesting the start of a new run, and is copied into ccBlock once the run starts.

`CmdPkt_Load_1D_Block` *`winBlock`*: This is the active 1D Window Parameter Block, if one is being used.

`CmdPkt_Load_1D_Block` *`stagedWinBlock`*: This is the window parameter block to use for the next run. It is setup by the handler which requested the new run, and is copied into winBlock once the run starts.

`Boolean` *`hasWindows`*: This flag indicates whether or not the active parameter block is using window filters.

`CcdId` *`fepCcd`*[6]: This is an array of CCD identifiers extracted from the active parameter block. The array is indexed by FEP Id.

`Boolean` *`fepVideo`*[6]: This is an array of FEP video gain selection, indexed by FEP Id and extracted from the active parameter block. BoolFalse indicates that the FEP's CCD should be clocked using 1electron/ADU. BoolTrue indicates that the CCD should be clocked to obtain 4 electrons/ADU.

`int` *`fepThresh`*[6][4]: This is an array of FEP quadrant threshold settings. The settings are indexed by FEP Id and video chain.

`unsigned` *`fepSplit`*[6][4]: This is an array of split threshold settings, indexed by FEP. These are copied from the active parameter block.

`Boolean` *`biasOk`*: This indicates if the bias maps are intact. If *`BoolTrue`*, then all configured bias maps are ready. If *`BoolFalse`*, then one or more bias maps need to be built.

Concurrency: Synchronous

Persistence: Persistent

37.6.1 SmContClocking()

Public member of: **`SmContClocking`**

Documentation:

This is the constructor for the **`SmContClocking`** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.6.2 activateParameters()

Public member of: **`SmContClocking`**

Return Class: **`void`**

Documentation:

This function copies the parameter blocks staged in *`stagedCcBlock`* and *`stagedWinBlock`* into *`ccBlock`* and *`winBlock`*.

Semantics:

Get the buffer address of the staged and main blocks, and get the length of the staged block and copy the data from the staged block into the main block. Then check the block if a window is specified, and if so, copy the staged window block into the active window block. Using the now active block, cache the FEP to CCD selections, FEP video selections and FEP threshold and split threshold selections into *`fepCcd`*, *`fepVideo`*, *`fepThresh`*, and *`fepSplit`*, respectively. Then iterate through the FEP CCD selections and verify that the corresponding DEA and FEP boards have power. If a DEA board or FEP board isn't powered, invalidate the corresponding FEP's CCD selection to prevent the run from using the FEP.

Concurrency: Guarded

37.6.3 checkBlock()

Public member of: **SmContClocking**

Return Class: **Boolean**

Arguments:

unsigned *blockid*

Documentation:

This function checks the integrity of the Continuous Clocking parameter block, indicated by *blockid*. If the parameter block is intact, the function returns *BoolTrue*. If the block has been corrupted (or contains invalid parameters TBD), the function returns *BoolFalse*.

Concurrency: Synchronous

37.6.4 dumpParameters()

Public member of: **SmContClocking**

Return Class: **Boolean**

Documentation:

This function posts the contents of the active Continuous Clocking and 1D window parameters to telemetry. If successful, the function returns *BoolTrue*. If the run was aborted, it returns *BoolFalse*.

Semantics:

Declare a Continuous Clocking Parameter block form, and use waitForPkt() to obtain a telemetry packet buffer for the form. If the run is aborted while waiting for the buffer, return *BoolFalse*. Once a buffer has been obtained, copy the active Continuous Clocking Parameter Block into the buffer. If a window list is used, append the active 1D Window list to the buffer. Then post the buffer for transfer to telemetry.

Concurrency: Guarded

37.6.5 getBlockIds()

Protected member of: **SmContClocking**

Return Class: **void**

Arguments:

unsigned& *blockId*
unsigned& *winId*

Documentation:

This function retrieves the parameter block ids from the Continuous Clocking Parameter block and the 1D Window parameter block (if used). On return, *blockId* will contain the parameter block id from the active Continuous Clocking Parameter Block. If a 1D window is used, *winId* will contain its parameter block id. If not, *winId* will contain 0xffffffff.

Concurrency: Guarded

37.6.6 getFepRequest()

Protected member of: **SmContClocking**

Return Class: **int**

Documentation:

This function returns the FEP request code used to start Continuous Clocking science data processing.

Concurrency: Guarded

37.6.7 loadBadMaps()

Public member of: **SmContClocking**

Return Class: **void**

Documentation:

This function loads the bad pixels and Continuous Clocking bad columns into their respective FEP bias maps.

Concurrency: Guarded

37.6.8 requiresBias()

Public member of: **SmContClocking**

Return Class: **Boolean**

Documentation:

This function determines if a bias computation is required. It first checks the parameter block for a bias request. If a bias is not mandated by the parameter block, it then queries each configured FEP to ensure that each has a valid bias. If either the parameter block mandates a bias computation, or if a FEP has an invalid bias, the function returns *BoolTrue*. If no bias is required by the configured mode, then it returns *BoolFalse*.

Concurrency: Guarded

37.6.9 setupDea()

Protected member of: **SmContClocking**

Return Class: **Boolean**

Documentation:

This function loads the Continuous Clocking SRAM image into the DEA CCD-controllers, and uses the PRAM Builder to build and load the Continuous Clocking clocking operations into the CCD-controller's PRAM. If successful, this function returns *BoolTrue*. If none of the controllers were loaded, the function returns *BoolFalse*.

Semantics:

If the run has a custom DEA load block, use *deaManager*.loadSequencers() to load it into the CCD controllers. If not, iterate through each configured FEP. Load the SRAM library using *sramLibrary*.load(), configure the PRAM builder and build and load the PRAM using *pramCc*.configure() and *pramCc*.build().

Concurrency: Guarded

37.6.10 setupEventProcess()

Protected member of: **SmContClocking**

Return Class: **void**

Arguments:

FepId *fep*
PmEvent& *process*

Documentation:

This function configures the event process, indicated by *process*, to handle events produced by the FEP, *fep*. It configures the run-header information, CCD geometry, sets up the filters, installs the filters, and installs process in the mode's CCD processor pointer array.

Concurrency: Guarded

37.6.11 setupFaint1x3()

Protected member of: **SmContClocking**

Return Class: **void**

Documentation:

This function sets up for Faint 1x3 event processing. This function iterates through each configured FEP, passing a separate **PmCcFaint1x3** instance to `setupEventProcess`() for each FEP being used.

Concurrency: Guarded

37.6.12 setupFep()

Protected member of: **SmContClocking**

Return Class: **Boolean**

Documentation:

This function loads code into each of the configured FEPs, and sets up the Continuous Clocking mode parameters into each. If successful, the function returns *`BoolTrue`*. If no FEPs were successfully configured, the function returns *`BoolFalse`*.

Semantics:

For each configured FEP, first load and run any customized code into the FEP, using *`fepManager`*.`loadRunProgram`(). Then query the status of the FEP, using *`fepManager`*.`queryFepStatus`(). If the query fails, then the FEP has crashed. Restart the FEP using the default program load. Use `setupFepBlock`() to build the BEP to FEP parameter block, and use *`fepManager`*.`configureFep`() to load the configuration into the FEP.

Concurrency: Guarded

37.6.13 setupFepBlock()

Protected member of: **SmContClocking**

Return Class: **Boolean**

Arguments:

FepId *fep*
FEPparmBlock& *fepblock*

Documentation:

This function sets up a Front End Processor Parameter Block, *fepblock*, for the FEP specified by *fep*. If successful, the function returns *BoolTrue*. If the FEP did not respond, it returns *BoolFalse*.

Concurrency: Guarded

37.6.14 setupGradeFilter()

Protected member of: **SmContClocking**

Return Class: **void**

Arguments:

FepId *fep*
FilterGrade& *grade*

Documentation:

This function sets up the Grade filter, *grade*. The passed FEP identifier, *fep*, is a hook to support enhancements which make the grade selection FEP-specific.

Concurrency: Guarded

37.6.15 setupGraded()

Protected member of: **SmContClocking**

Return Class: **void**

Documentation:

This function sets up for Graded event processing.This function iterates through each configured FEP, passing a separate **PmCcGraded** instance to `setupEventProcess`() for each FEP being used.

Concurrency: Guarded

37.6.16 setupPhFilter()

Protected member of: **SmContClocking**

Return Class: **void**

Arguments:

FepId *fep*
FilterPh& *phFilter*

Documentation:

This function configures the Pulse Height filter, *phFilter*. The passed FEP identifier, *fep*, is a hook to support enhancements which make the grade selection FEP-specific.

Concurrency: Guarded

37.6.17 setupProcess()

Protected member of: **SmContClocking**

Return Class: **void**

Documentation:

This function sets up the processing objects used to process data for the run.

Semantics:

This function examines the configured FEP and BEP processing mode selections. If performing `raw` mode, call `setupRaw`(). If performing 1x3 Faint Mode, call `setupFaint1x3`(), and if performing Graded Mode, call `setupGraded`().

Concurrency: Guarded

37.6.18 setupRaw()

Protected member of: **SmContClocking**

Return Class: **void**

Documentation:

This function sets up for Raw data processing. For each FEP in use, it sets the mode, the CCD Id, geometry, into a separate **PmCcRaw** instance. It also sets up a distinct window list filter and assigns it to the **PmCcRaw** instance. It then passes the instance to `assignFepProcess`() to associate the instance with the FEP.

Concurrency: Guarded

37.6.19 setupWindowFilter()

Protected member of: **SmContClocking**

Return Class: **void**

Arguments:

FepId *fep*
FilterWindow& *window*

Documentation:

This function sets up the window list filter, *window*. The passed FEP identifier, *fep*, is a hook to support enhancements which make the grade selection FEP-specific.

Concurrency: Guarded

37.6.20 stageParameters()

Public member of: **SmContClocking**

Return Class: **void**

Arguments:

unsigned *blockid*

Documentation:

This function fetches the Continuous Clocking parameter block, indicated by *blockid*, and stores its contents into *stagedCcBlock*. If a window block is referenced within the block, it is fetched and stored into *stagedWinBlock*.

Concurrency: Synchronous

37.6.21 terminate()

Public member of: **SmContClocking**

Return Class: **void**

Documentation:

This function cleans up after a Continuous Clocking science run. It produces and posts a science run report.

Concurrency: Guarded

37.6.22 useBiasThief()

Public member of: **SmContClocking**

Return Class: **Boolean**

Documentation:

This function determines whether or not the Bias Thief should be invoked. It returns *BoolTrue* if the bias thief should send the pixel bias values, and *BoolFalse* if not.

Concurrency: Guarded

37.7 Class EventExposure

Documentation:

This class maintains exposure and some science run geometry information, including the scale factors resulting from on-chip pixel summing, the row offset used for sub-array readout, the configured split threshold values, the delta overclock values produced by the FEPs, etc.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **none**

Implementation Uses:

FEPexpEndRec
FEPexpRec

Public Interface:

Operations:

```
EventExposure()
copyExpEnd()
copyExpStart()
getExposureNumber()
getGeometry()
getOverclockBase()
getOverclockDelta()
getQuadrant()
getSplitThreshhold()
getThresholdCnt()
mapPosition()
setGeometry()
setSplitThreshold()
```

Private Interface:

Has-A Relationships:

unsigned *grmul*: This contains a copy of the row scale factor supplied by `setGeometry`. This factor is the result of on-chip pixel row summing.

unsigned *gcmul*: This is a copy of the column scale factor supplied by `setGeometry`. This factor is the result of on-chip pixel column summing.

unsigned *groff*: This is a copy of the CCD row offset, supplied by `setGeometry`. This factor is a result of sub-array readout clocking of the CCD.

Boolean *gcpQuad*: This flag indicates whether or not to replicate the quadrant overclock values supplied by the FEP. If `setGeometry` indicates use of all four output nodes, this value is *BoolFalse*. If it indicates use of only two output nodes, this flag is set to *BoolTrue*, and overclock 0 reported by the FEP is copied into slots for nodes A and B, and overclock 1 is copied into slots for nodes C and D.

unsigned *split*[4]: This array contains the configured split thresholds for the run.

unsigned *expNum*: This variable contains the last reported exposure number.

unsigned *expOcBase*[4]: This array contains the base overclock values for the run, reported from the FEP.

int *expOcDelta*[4]: This array contains the last reported delta-overclock values, reported from the FEP.

unsigned *expFepTime*: This variable contains the last reported FEP timestamp value

unsigned *expThresholdCnt*: This variable contains the count of pixels above threshold last reported by the FEP.

Concurrency: Guarded

Persistence: Transient

37.7.1 EventExposure()

Public member of: **`EventExposure`**

Documentation:

This is the constructor for the **`EventExposure`** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.7.2 copyExpEnd()

Public member of: **`EventExposure`**

Return Class: **`void`**

Arguments:

`const FEPexpEndRec*` *`dataptr`*

Documentation:

This function copies the number of pixels which were above threshold in the completed exposure, from the FEP produced Exposure End record into *`expThresholdCnt`*. The passed *`dataptr`* points to the FEP produced Exposure End record.

Preconditions:

If exposure start record has not yet been processed using `copyExpStart`, or if *`expNum`* does not match the exposure number in the Exposure End record, this function will report the occurrence to software housekeeping (statistic codes TBD).

Concurrency: Guarded

37.7.3 copyExpStart()

Public member of: **`EventExposure`**

Return Class: **`void`**

Arguments:

`const FEPexpRec*` *`dataptr`*

Documentation:

This function respectively copies the exposure number, overclock base values, overclock delta values, and FEP timestamp, from the FEP-produced Exposure Start Record pointed to by *`dataptr`*, into *`expNum`*, *`expOcBase`*, *`expOcDelta`*, and *`expFepTime`*. If an exposure has been started, but an End Record has not been processed, the occurrence is reported to software housekeeping, and the uncompleted exposure's information is overwritten.

Concurrency: Guarded

37.7.4 getExposureNumber()

Public member of: **`EventExposure`**

Return Class: **`unsigned`**

Documentation:

This function returns the most recently reported exposure number, *`expNum`*. An exposure start record must have been processed using `copyExpStart`, otherwise this function returns the unlikely exposure number 0xffffffff.

Concurrency: Synchronous

37.7.5 getGeometry()

Public member of: **EventExposure**

Return Class: **void**

Arguments:

unsigned& *rowscale*
unsigned& *colscale*
unsigned& *rowoffset*
QuadMode& *nodeSelect*

Documentation:

This function returns the exposure geometry values. Upon return, *rowscale* and *colscale* contain the row and column scaling factors, *rowoffset* contains the CCD row offset, and *nodeSelect* contains the output node mode selection (Full,Diag,AC,BD).

Concurrency: Guarded

37.7.6 getOverclockBase()

Public member of: **EventExposure**

Return Class: **unsigned**

Arguments:

QuadCode *node*

Documentation:

This function returns the overclock base value for the output node specified by *node*. An exposure start record must have been processed via copyExpStart, otherwise this function returns 0xffffffff.

Concurrency: Synchronous

37.7.7 getOverclockDelta()

Public member of: **`EventExposure`**

Return Class: **`int`**

Arguments:

`QuadCode` *`node`*

Documentation:

This function returns the signed delta-overclock value computed for the CCD output node indexed by *`node`*. An exposure start record must have been received, otherwise this function returns 0x80000000.

Concurrency: Synchronous

37.7.8 getQuadrant()

Public member of: **`EventExposure`**

Return Class: **`QuadCode`**

Arguments:

`unsigned` *`pixcol`*

Documentation:

This function maps the pixel column, *`pixcol`*, to an output node and returns the node identifier.

Preconditions:

`setGeometry` must have been called.

Concurrency: Synchronous

37.7.9 getSplitThreshhold()

Public member of: **`EventExposure`**

Return Class: **`unsigned`**

Arguments:

`QuadCode` *node*

Documentation:

This function returns the split threshold associated with the output node, *node*.

Preconditions:

`setSplitThreshold` must have been called.

Concurrency: Synchronous

37.7.10 getThresholdCnt()

Public member of: **`EventExposure`**

Return Class: **`unsigned`**

Documentation:

This function returns the number of pixels above threshold in the last processed exposure. An exposure end must have been processed, using `copyExpEnd`, otherwise this function returns 0xffffffff.

Concurrency: Synchronous

37.7.11 mapPosition()

Public member of: **EventExposure**

Return Class: **void**

Arguments:

unsigned *pixrow*
unsigned *pixcol*
unsigned& *ccdrow*
unsigned& *ccdcol*

Documentation:

This function maps the clocked pixel position into CCD coordinates. *pixrow* and *pixcol* are the clocked pixel row and column values. Upon returning, this function sets *ccdrow* and *ccdcol* to the corresponding CCD row and column position.

Preconditions:

`setGeometry` must have been called.

Concurrency: Synchronous

37.7.12 setGeometry()

Public member of: **EventExposure**

Return Class: **void**

Arguments:

unsigned *rowscale*
unsigned *colscale*
unsigned *rowoffset*
QuadMode *quadMode*

Documentation:

This function stores the run geometry information into this instance, and uses *quadMode* to determine how to handle overclocks reported by the FEP. *rowscale* and *colscale* are the row and column scale factors for the run. *rowoffset* is the starting offset of the first row clocked out of the CCD. *quadMode* is the quadrant selection for the run (Full, Diag, AC, or BD).

Concurrency: Guarded

37.7.13 setSplitThreshold()

Public member of: **EventExposure**

Return Class: **void**

Arguments:

unsigned *nodeA*
unsigned *nodeB*
unsigned *nodeC*
unsigned *nodeD*

Documentation:

This function sets the split threshold for each of the CCD output nodes. *nodeA*, *nodeB*, *nodeC*, and *nodeD* are the threshold values for nodes A, B, C and D respectively. If only two nodes are in use for the run, the values for A and C should be the same, and the values for B and D should be the same.

Concurrency: Guarded

37.8 Class PhHistogram

Documentation:

This class represents a histogram of pulse heights from one Front End Processor. This class is used to access the histogram data during Back End data processing and telemetry production.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

FEPeventRecHist

Public Interface:

Operations:

```
copyHeader()
getExposureCount()
getExposureEnd()
getExposureStart()
getOverclockMax()
getOverclockMean()
getOverclockMin()
getOverclockVariance()
```

Private Interface:

Has-A Relationships:

unsigned *ocMean[4]*: This is a copy of the histogram's overclock mean values.

unsigned *ocMin[4]*: This is a copy of the minimum overclock levels from each quadrant.

unsigned *ocVar[4]*: This is a copy of the overclock variance levels from each quadrant.

unsigned *ocMax[4]*: These are the maximum overclocks from each quadrant.

unsigned *expStart*: This is the starting exposure number used to compute the histogram.

unsigned *expEnd*: This is the number of the last exposure used to build the histogram.

Concurrency: Guarded

Persistence: Transient

37.8.1 copyHeader()

Public member of: **PhHistogram**

Return Class: **void**

Arguments:

const FEPeventRecHist* *record*
QuadMode *mode*

Documentation:

This function copies the header information into all of its state variables from the histogram record, *record*. *mode* indicates which quadrants were used to build the histograms.

Concurrency: Guarded

37.8.2 getExposureCount()

Public member of: **PhHistogram**

Return Class: **unsigned**

Documentation:

This function returns the number of exposures actually integrated in the histogram.

NOTE: Currently, this information is not provided in the FEP to BEP histogram record, and this function returns 0. Either add the info to the record, or add a function to set the configured count and assume all exposures were included.

Concurrency: Guarded

37.8.3 getExposureEnd()

Public member of: **PhHistogram**

Return Class: **unsigned**

Documentation:

This function returns the exposure number of the last exposure added to the histogram, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.8.4 getExposureStart()

Public member of: **PhHistogram**

Return Class: **unsigned**

Documentation:

This function returns the exposure number of the first exposure accumulated into the histogram, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.8.5 getOverclockMax()

Public member of: **PhHistogram**

Return Class: **unsigned**

Arguments:

QuadCode *quadrant*

Documentation:

This function returns the maximum overclock level detected from quadrant, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.8.6 getOverclockMean()

Public member of: **PhHistogram**

Return Class: **unsigned**

Arguments:

QuadCode *quadrant*

Documentation:

This function returns the mean overclock level from quadrant, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.8.7 getOverclockMin()

Public member of: **PhHistogram**

Return Class: **unsigned**

Arguments:

QuadCode *quadrant*

Documentation:

This function returns the minimum overclock level detected from quadrant, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.8.8 getOverclockVariance()

Public member of: **PhHistogram**

Return Class: **unsigned**

Arguments:

QuadCode *quadrant*

Documentation:

This function returns the variance of the overclocks from quadrant, as specified in the FEP to BEP histogram record, **FEPeventRecHist**.

Concurrency: Guarded

37.9 Class PixelRow

Documentation:

This class represents a row of raw pixel pulse heights produced by one Front End Processor. This class provides functions to load new pixel data, flag regions to be telemetered or discarded, and to access the pixels within the row.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

EventExposure

Public Interface:

Operations:
`acceptRegion()`
`attachData()`
`discardRegion()`
`getNextSendRegion()`
`getOverclockPtr()`
`getPixelPtr()`
`getRange()`
`setup()`

Protected Interface:

Operations: `flagRegion()`

Private Interface:

Has-A Relationships:

unsigned char *sendMask*[1024]: This is an array of flags indicating the filtered state of a given pixel in the row. Each entry corresponds to one pixel. The codes for this array are not yet assigned, but will reflect at least the following states: Pixel Accepted, Pixel Rejected.

FEPeventRecRaw* *recPtr*: This is a pointer to the raw data record.

unsigned *ccdRow*: This is the CCD row position of the raw pixels.

unsigned *rmul*: This is the row scale factor.

unsigned *cmul*: This is the column scaling factor.

unsigned *maskLimit*: this is the scaled mask limit for the row.

unsigned *subarrayStart*: This is the starting CCD row number for the image.

Concurrency: Guarded

Persistence: Transient

37.9.1 acceptRegion()

Public member of: **PixelRow**

Return Class: **unsigned**

Arguments:

unsigned *mincol*
unsigned *maxcol*

Documentation:

This function flags the pixels between *mincol* and *maxcol* to be sent to telemetry. *mincol* and *maxcol* are expressed in CCD coordinates, rounded up to the nearest clocked pixel. This function returns the number of affected pixels.

Concurrency: Guarded

37.9.2 attachData()

Public member of: **PixelRow**

Return Class: **void**

Arguments:

const FEPeventRecRaw* *dataptr*

Documentation:

This function sets its internal data pointer to the raw record pointed to by *dataptr*, extracts the row position, converts it to absolute CCD coordinates, and stores the position in *ccdrow*.

Concurrency: Guarded

37.9.3 discardRegion()

Public member of: **PixelRow**

Return Class: **unsigned**

Arguments:

unsigned *mincol*
unsigned *maxcol*

Documentation:

This function flags the CCD pixels from *mincol* to *maxcol* so they are not sent into telemetry. *mincol* and *maxcol* are expressed in CCD coordinates. This function returns the number of affected pixels.

Concurrency: Guarded

37.9.4 flagRegion()

Protected member of: **PixelRow**

Return Class: **unsigned**

Arguments:

unsigned *mincol*
unsigned *maxcol*
unsigned char *flag*

Documentation:

This function stores *flag* into the *sendMask*[] array slots corresponding to the CCD pixels from *mincol* to *maxcol* so they are not sent into telemetry. *mincol* and *maxcol* are expressed in CCD coordinates. This function returns the number of affected pixels.

Concurrency: Guarded

37.9.5 getNextSendRegion()

Public member of: **PixelRow**

Return Class: **void**

Arguments:

unsigned& *start*
unsigned& *count*

Documentation:

This function obtains the next region of the pixel row to be sent to telemetry. On input, *start* is the starting pixel column position of the last region returned by this function or sent, and *count* contains the previous region's length. If *start* and *count* are zero then this is the first call for the row. Upon returning, *start* contains the next region of the row to send and *count* contains the number of pixels to send in the region.

Semantics:

Adjust *start* by adding *count*. Scan *sendMask* until a non-discard code is reached and set *start* to that point. Continue scanning until a discard code is reached, and set *count* to the number of pixels in the region.

Concurrency: Guarded

37.9.6 getOverclockPtr()

Public member of: **PixelRow**

Return Class: **const unsigned short***

Documentation:

This function returns a pointer to the overclock pixels in the row (i.e. the address of *recPtr->oc*).

Concurrency: Guarded

37.9.7 getPixelPtr()

Public member of: **PixelRow**

Return Class: **const unsigned short***

Arguments:

unsigned *pixcol*

Documentation:

This function returns a pointer to the pixels in the row starting from pixel column *pixcol* (i.e. the address of *recPtr*->*p*[*pixcol*]).

Concurrency: Guarded

37.9.8 getRange()

Public member of: **PixelRow**

Return Class: **void**

Arguments:

unsigned& *row*
unsigned& *mincol*
unsigned& *maxcol*

Documentation:

This function returns the CCD position of the row of raw pixels. It sets row to the CCD row position of the pixels, and *mincol* and *maxcol* to the minimum and maximum column positions represented by the array.

Concurrency: Guarded

37.9.9 setup()

Public member of: **PixelRow**

Return Class: **void**

Arguments:

const EventExposure* *exposure*

Documentation:

This function sets up the pixel row, using the information in *exposure*.

Semantics:

Get the scaling factors, subarray start, and node selection using *exposure*->getGeometry(), storing the results in *rmul*, *cmul*, *subarrayStart* and *nodeSelect*. The set the sendMask's limit, *maskLimit*, by dividing the maximum number of columns in a row (always 1024 for ACIS) by the column summing factor, *cmul*.

Concurrency: Guarded

37.10 Class PixelEvent

Documentation:

This abstract class represents an candidate X-ray event, and provides functions to obtain the position, the grade code and the pulse height of the event.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Implementation Uses:

EventExposure

Public Interface:

Operations:
```
PixelEvent()
getCcdPosition()
getGrade()
getPulseHeight()
```

Protected Interface:

Has-A Relationships:

unsigned *vPh*: This variable contains the pulse height of the event.

GradeCode *vGrade*: This variable contains the grade of the event.

Operations:
```
setup()
correctPixelPh()
```

Private Interface:

Has-A Relationships:

unsigned *ccdrow*: This variable contains the event's row position, in CCD-coordinates.

unsigned *ccdcol*: This value contains the column position of the event, in CCD-coordinates.

Concurrency: Guarded

Persistence: Transient

37.10.1 PixelEvent()

Public member of: **`PixelEvent`**

Documentation:

This is the constructor for the **`PixelEvent`** class. This function zeros the instance variables of the constructed instance.

Concurrency: Synchronous

37.10.2 correctPixelPh()

Protected member of: **`PixelEvent`**

Return Class: **`int`**

Arguments:

`unsigned` *`pixelPh`*
`unsigned` *`biasPh`*
`int` *`deltaOc`*

Documentation:

This function computes the corrected pixel pulse height. *`pixelPh`* is the raw pixel pulse height, *`biasPh`* is the bias value associated with the pixel, and *`deltaOc`* is the delta overclock value associated with the pixel. If the bias value is invalid, or if *`deltaOc`* is too small, this function will return a negative value.

Concurrency: Synchronous

37.10.3 getCcdPosition()

Public member of: **PixelEvent**

Return Class: **void**

Arguments:

unsigned& *rowOut*
unsigned& *colOut*

Documentation:

This function returns the CCD pixel position of the center of the event. *rowOut* is the row (or continuous clocking row count) of the center of the event, and *colOut* is set to the column position of the center of the event. If the mode was summing rows or columns, the position is scaled to CCD coordinates, but represents the bottom-left corner of the summed pixel grid.

Concurrency: Guarded

37.10.4 getGrade()

Public member of: **PixelEvent**

Return Class: **GradeCode**

Documentation:

This function returns the grade code of the 3x3 event.

Concurrency: Guarded

37.10.5 getPulseHeight()

Public member of: **PixelEvent**

Return Class: **unsigned**

Documentation:

This function returns the total pulse height (i.e. event amplitude) of a 3x3 event.

Concurrency: Guarded

37.10.6 setup()

Protected member of: **PixelEvent**

Return Class: **void**

Arguments:

const ExposureInfo* *exposure*
unsigned *pixrow*
unsigned *pixcol*

Documentation:

This function resets the state of any data dependent internally computed values. *exposure* points to an exposure record, used to map pixel addresses into CCD positions. *pixrow* and *pixcol* are the row and column position of the event in pixel coordinates.

Concurrency: Guarded

37.11 Class Pixel1x3

Documentation:

This class represents a candidate event produced when in Continuous Clocking Mode, represented as a single row of three adjacent pixels. This class provides functions to load new pixel data, to obtain the bias value associated with a pixel within the event, and to obtain the uncorrected pulse-height of a pixel within the event.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`PixelEvent`**

Public Uses:

`EventExposure`
`FEPeventRec1x3`

Public Interface:

Operations:
`Pixel1x3()`
`attachData()`
`getBias()`
`getPixel()`

Protected Interface:

Operations: `computePhGrade()`

Private Interface:

Has-A Relationships:

`const FEPeventRec1x3`* *`dataPtr`*: This is a pointer to the 1x3 event record associated with the instance via `attachData`().

`unsigned` *`pixcol`*: This is the pixel column position of the event.

`unsigned` *`pixrow`*: This is the pixel row position of the center of the event, relative to the start of the 512 row continuous clocking data set.

Concurrency: Guarded

Persistence: Transient

37.11.1 Pixel1x3()

Public member of: **`Pixel1x3`**

Documentation:

This is the constructor for the **`Pixel1x3`** class. This function invokes the parent constructor, `PixelEvent`(), to zero the instance variables of the constructed instance.

Concurrency: Synchronous

37.11.2 attachData()

Public member of: **`Pixel1x3`**

Return Class: **`void`**

Arguments:

`FEPevent1x3*` *`dataPtr`*
`EventExposure*` *`exposure`*

Documentation:

This function stores the reference to the FEP record pointed to by *`dataPtr`*. This reference is used to obtain the position, pixel and bias information from the FEP record.

Semantics:

The function uses the `setup`() function to register the CCD row and column position of the event. It then uses *`exposure`*->`getQuadrant`() to determine which quadrant each column position belongs to, and then uses *`exposure`*->`getOverclockDelta`() and *`exposure`*->`getSplitThreshold`() to obtain the delta-overclock levels and split threshold levels for each column. The function then uses `computePhGrade`() to compute and store the pulse height and spatial distribution grade of the event, passing the overclock and threshold information provided by *`exposure`*.

Concurrency: Guarded

37.11.3 computePhGrade()

Protected member of: **Pixel1x3**

Return Class: **void**

Arguments:

int *doclk*[3]
unsigned *split*[2]

Documentation:

This function computes and stores the pulse height and spatial distribution grade of the event. *doclk* are the delta-overclock values of the left, center, and right columns of the event. *split* contains the split thresholds for the left and right columns (center split threshold is not used for a 1x3 event).

Semantics:

This function zeros the *vPh* and *vGrade* instance variables. It then uses **PixelEvent**::correctPixelPh() to get the corrected pulse height of the center pixel and stores the result in *vPh*. It then uses **PixelEvent**::correctPixelPh() to get the corrected pulse heights of the left and right events, and compares each against their respective split threshold. If they are greater than or equal to the split threshold, their corrected pulse height is added to *vPh*, and the grade bit corresponding to the column is added to *vGrade*.

Concurrency: Guarded

37.11.4 getBias()

Public member of: **Pixel1x3**

Return Class: **unsigned**

Arguments:

unsigned *column*

Documentation:

This function returns the pixel bias value identified by *column*. The argument *column* must have one of the following values: 0, 1, or 2.

Concurrency: Guarded

37.11.5 getPixel()

Public member of: **Pixel1x3**

Return Class: **unsigned**

Arguments:

unsigned *column*

Documentation:

This function returns the raw pixel pulse height value identified by *column*. *column* must have one of the following values: 0, 1, or 2.

Concurrency: Guarded

37.12 Class Pixel3x3

Documentation:

This class represents a 3x3 event, produced by a Front End Processor in Timed Exposure Mode. This class provides functions to load new pixel data, to obtain the bias value associated with a pixel within the event, and to obtain the uncorrected pulse-height of a pixel within the event.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`PixelEvent`**

Public Uses:

`FEPeventRec3x3`

Public Interface:

Operations:
```
Pixel3x3()
attachData()
getBias()
getPixel()
```

Protected Interface:

Operations:
```
computePhGrade()
gradeCornerPixel()
gradeEdgePixel()
```

Private Interface:

Has-A Relationships:.

`const FEPeventRec3x3`* *`dataPtr`*: This is a pointer to the 3x3 event record associated with the instance via `attachData`().

`unsigned` *`pixrow`*: This is the pixel row of the center of the event.

`unsigned` *`pixcol`*: This is the pixel column of the center of the event.

Concurrency: Guarded

Persistence: Transient

37.12.1 Pixel3x3()

Public member of: **Pixel3x3**

Documentation:

This is the constructor for the **Pixel3x3** class. This function invokes the parent constructor, `PixelEvent`(), to zero the instance variables of the constructed instance.

Concurrency: Synchronous

37.12.2 attachData()

Public member of: **Pixel3x3**

Return Class: **void**

Arguments:

FEPeventRecord3x3* *dataptr*
EventExposure* *exposure*

Documentation:

This function copies the event information contained in the record pointed to by *dataptr*, and computes the CCD position of the event, the pulse height of the event, and the event grade, using geometry, overclock and threshold information provided by *exposure*.

Semantics:

The function uses the `setup`() function to register the CCD row and column position of the event. It then uses *exposure*->`getQuadrant`() to determine which quadrant each column position belongs to, and then uses *exposure*->`getOverclockDelta`() and *exposure*->`getSplitThreshold`() to obtain the delta-overclock levels and split threshold levels for each column. The function then uses `computePhGrade`() to compute and store the pulse height and spatial distribution grade of the event, passing the overclock and threshold information provided by *exposure*.

Concurrency: Guarded

37.12.3 computePhGrade()

Protected member of: **Pixel3x3**

Return Class: **void**

Arguments:

int *doclk*[3]
unsigned *split*[3]

Documentation:

This function computes the pulse height and grade of the 3x3 event. *doclk* contains the delta-overclock values for the left, center, and right columns of the event. *split* contains the split threshold values for the corresponding columns.

Semantics:

Zeros the grade, **PixelEvent**::*vGrade*, set the pulse height, **PixelEvent**::*vPh*, to the corrected pulse height of the center pixel (**PixelEvent**::computePixelPh()). Compute the corrected pulse heights for each of the edge pixels and process the result using gradeEdgePixel(). Then compute the corrected pulse heights for each of the corner pixels and figure their contribution using gradeCornerPixel().

Concurrency: Guarded

37.12.4 getBias()

Public member of: **Pixel3x3**

Return Class: **unsigned**

Arguments:

unsigned *row*
unsigned *col*

Documentation:

This function returns the pixel bias located at [*row*][*col*], where *row* and *col* are either 0, 1 or 2.

Concurrency: Guarded

37.12.5 getPixel()

Public member of: **Pixel3x3**

Return Class: **unsigned**

Arguments:

unsigned *row*
unsigned *col*

Documentation:

This function returns the pulse height of the pixel located at [*row*][*col*], where *row* and *col* are either 0, 1 or 2.

Concurrency: Guarded

37.12.6 gradeCornerPixel()

Protected member of: **Pixel3x3**

Return Class: **void**

Arguments:

int *corrected*
unsigned *threshold*
unsigned *gradeBit*
Boolean *phEnabled*

Documentation:

This function compares the corrected pixel pulse height, *corrected*, with the split threshold, *threshold*. If *corrected* is greater than *threshold*, the function sets the bit indexed by *gradeBit*, into the event's grade code, **PixelEvent**::*vGrade*. Also, if *phEnabled* is *BoolTrue* (i.e. an adjacent edge pixel is also above split threshold), it adds the corrected pulse height to the total pulse height of the event, **PixelEvent**::*vPh*.

Concurrency: Guarded

37.12.7 gradeEdgePixel()

Protected member of: **Pixel3x3**

Return Class: **void**

Arguments:

int *corrected*
unsigned *threshold*
unsigned *gradeBit*
Boolean& *corner1*
Boolean& *corner2*

Documentation:

This function compares the corrected pixel pulse height, *corrected*, against the split threshold, *threshold*. If corrected is greater than or equal to *threshold*, the function sets the bit, indexed by *gradeBit*, in the event's grade code, **PixelEvent**::*vGrade*, adds the corrected pulse height to the total for the event, **PixelEvent**::*vPh*, and sets the two corner pixel flags, *corner1* and *corner2*, to *BoolTrue*, indicating that either pixel may be considered as part of the event.

Concurrency: Guarded

37.13 Class Filter

Documentation:

This class is an abstract class which represents a data filter residing on the Back End Processor. This class provides functions to accept and reject data items, and to maintain counts of the accepted and rejected items.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
Filter()
accept()
getAcceptCnt()
getDiscardCnt()
reject()
resetCounters()
```

Private Interface:

Has-A Relationships:

unsigned *accepted*: This variable contains the number of items accepted by the filter since the most recent call to `resetCounters`. This counter is zeroed by `resetCounters`, typically once per exposure, and advanced during exposure processing by calls to `accept`.

unsigned *rejected*: This variable contains the number of items rejected by the filter since the most recent call to `resetCounters`. This counter is zeroed by `resetCounters`, typically once per exposure, and is advanced during exposure processing by calls to `reject`.

Concurrency: Guarded

Persistence: Persistent

37.13.1 Filter()

Public member of: **`Filter`**

Documentation:

This is the constructor for the **`Filter`** class. This function zeros the instances *`accepted`* and *`rejected`* counters.

Concurrency: Guarded

37.13.2 accept()

Public member of: **`Filter`**

Return Class: **`void`**

Arguments:

`unsigned` *`cnt`*

Documentation:

This function increments the filter's accept counter by an amount specified by *`cnt`*.

Concurrency: Guarded

37.13.3 getAcceptCnt()

Public member of: **`Filter`**

Return Class: **`unsigned`**

Documentation:

This function returns the number of events accepted by the filter since the most recent call to `resetCounters`.

Concurrency: Guarded

37.13.4 getDiscardCnt()

Public member of: **`Filter`**

Return Class: **unsigned**

Documentation:

This function returns the number of items rejected by the filter since the most recent call to `resetCounters`.

Concurrency: Guarded

37.13.5 reject()

Public member of: **Filter**

Return Class: **void**

Arguments:

unsigned *cnt*

Documentation:

This function increments the filter's discard counter by the amount specified by *cnt*.

Concurrency: Guarded

37.13.6 resetCounters()

Public member of: **Filter**

Return Class: **void**

Documentation:

This function resets the data element accept and reject counters to 0.

Concurrency: Guarded

37.14 Class FilterWindow

Documentation:

This class is responsible for filtering events and raw-mode pixel rows based on a collection of windows. This class provides functions to configure a new set of windows, to filter a row of raw pixel data based on the configured windows, and to filter an event on the configured windows.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`Filter`**

Public Uses:

`PixelEvent`
`PixelRow`

Public Interface:

Operations:
`FilterWindow()`
`addWindow()`
`filterEvent()`
`filterRow()`
`resetWindows()`

Private Interface:

Has-A Relationships:

`WindowInfo` *`windows`*[36]: This is an array of window definition structures. Each structure specifies the bounding box of a 2-D window, the sample limit and pulse height range for the window, and the current sample counter for the window. Each **`WindowInfo`** structure contains the following fields:

`unsigned` *`minRow`*: Minimum row position of the window

`unsigned` *`maxRow`*: Maximum row position of window

`unsigned` *`minCol`*: Minimum column position of window

`unsigned` *`maxCol`*: Maximum column position of window

`FilterPh` *`phFilter`*: Pulse height filter used by window

`unsigned` *`sample`*: Sample limit of window

`unsigned` *`eventCnt`*: Number of events currently in sample cycle

`unsigned` *`windowCnt`*: This variable specifies the number of windows being used for the filter.

Concurrency: Guarded

Persistence: Persistent

37.14.1 FilterWindow()

Public member of: **`FilterWindow`**

Documentation:

This is the constructor for the **`FilterWindow`** class. This function zeros the *`windowCnt`* instance variable.

Concurrency: Guarded

37.14.2 addWindow()

Public member of: **FilterWindow**

Return Class: **Boolean**

Arguments:

unsigned *minRow*
unsigned *minCol*
unsigned *nrows*
unsigned *ncols*
unsigned *sample*
unsigned *lowerPh*
unsigned *phRange*

Documentation:

This function adds a window to the filter set. *minRow* is the minimum CCD row position covered by the window, and *minCol* is the minimum CCD column covered. *nrows* is the total number of CCD rows covered by the window, and *ncols* is the number of columns. (NOTE: To configure a 1-d window, set *minRow* to 0 and *nrows* to 1024. If *nrows* or *ncols* is 0, then the window will not process any events. If *nrows* or *ncols* is greater than 1024, the offending value will be clipped, and set to 1024). When *sample* is greater than zero, its value minus 1 is the number of events to skip for each event produced by the window. If *sample* is 0, then all events are discarded by the window. *lowerPh* is the lowest event pulse height accepted by the window, and *phRange* is the number of pulse heights above *lowerPh* accepted by the window (i.e. an event must have pulse height greater than or equal to *lowerPh* and less than *lowerPh* + *phRange*). (NOTE: If *phRange* is 0, or *lowerPh* is greater than the maximum event pulse height (~73K), then all intersecting events will be rejected by the window).This function copies this information into the *windows* element indexed by *windowCnt*, and increments *windowCnt*. If there are no **WindowInfo** structures remaining at the time of the call, this function returns *BoolFalse*, otherwise it returns *BoolTrue*.

Concurrency: Guarded

37.14.3 resetWindows()

Public member of: **`FilterWindow`**

Return Class: **`void`**

Documentation:

This function resets the *`windowCnt`* index to 0, eliminating all windows added by `addWindow`().

Concurrency: Guarded

37.14.4 filterEvent()

Public member of: **`FilterWindow`**

Return Class: **`Boolean`**

Arguments:

`PixelEvent&` *`event`*

Documentation:

This function filters the pixel event referenced by *`event`*. If the event is accepted for further processing, this function returns *`BoolTrue`*. If the event is rejected, the function returns *`BoolFalse`*.

Semantics:

Get the position and pulse height of the event using *`event`*`.getCcdPosition`() and *`event`*`.getPulseHeight`(). For each window in *`windows`*[] (up to *`windowCnt`*), compare the position to the row and column ranges of the window. If the position is not within the limits of the window, check the next window. If no window handles the event, accept the event. If the position is within a window's limits, check the sample counter of the window. If the counter is at its limit, check the pulse height range of the window against the event. If the pulse height is in range, accept the event, and increment the sample counter. If the sample counter is not at its limit, or the pulse height is not within the acceptable range, reject the event.

Concurrency: Guarded

37.14.5 filterRow()

Public member of: **`FilterWindow`**

Return Class: **`Boolean`**

Arguments:

`PixelRow&` *`pixelRow`*

Documentation:

This function filters and clips the row of raw pixel pulse heights, referenced by *`pixelRow`*.

Semantics:

Initially, all pixels in *`pixelRow`* should be flagged to be sent to telemetry. For every window, working backwards in the *`windows`*[] array, determine the intersection of the row of pixels with a given window. If the row intersects the window, use the window's configured sample limit to determine if the window accepts or rejects the intersecting region. If the sample limit is not 0, then use *`pixelRow`*.`discardRegion`() to flag the intersecting pixels to be clipped out of the final telemetered image. If the sample limit is 0, then use *`pixelRow`*.`acceptRegion`() to flag the intersecting pixels to be sent. Subsequent windows may or may not override certain pixel flags with subsequent calls to `acceptRegion`() or `discardRegion`(). The last window which touches a given pixel determines the final send/don't send decision of that pixel.

Concurrency: Guarded

37.15 Class FilterGrade

Documentation:

This class is responsible for filtering events based on their grade code. This class handles grade codes which can range from 0 to 255. This range encompasses both Timed Exposure and Continuous Clocking grade code ranges and is used for both modes.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`Filter`**

Public Uses:

`PixelEvent`

Public Interface:

Operations:
`allow()`
`disableAll()`
`filterEvent()`

Private Interface:

Has-A Relationships:

`unsigned char` *`grades`*[32]: This array contains one bit-element for each possible grade code (in a 3x3 event), and is indexed by grade code. If an element is ‘0’, then events with the corresponding grade code are to be rejected. If the element is ‘1’, then the event is accepted for further processing.

Concurrency: Guarded

Persistence: Transient

37.15.1 allow()

Public member of: **`FilterGrade`**

Return Class: **`void`**

Arguments:

`GradeCode` *`code`*

Documentation:

This function causes the filter to accept events with the grade code specified by *`code`* (i.e. sets *`grade`*[*`code`*] to *`BoolTrue`*).

Concurrency: Guarded

37.15.2 disableAll()

Public member of: **`FilterGrade`**

Return Class: **`void`**

Documentation:

This function causes all grade codes to be rejected (i.e. sets the contents of the *`grade`* array to *`BoolFalse`*).

Concurrency: Guarded

37.15.3 filterEvent()

Public member of: **FilterGrade**

Return Class: **Boolean**

Arguments:

PixelEvent& *event*

Documentation:

This function filters an event based on its grade code. The function calls *event*.getGrade() to obtain the grade code of the event. If the indexed flag in *grade*[] is *BoolTrue*, the event is accepted and the function returns *BoolTrue*, else it is rejected and the function returns *BoolFalse*.

Concurrency: Guarded

37.16 Class FilterPh

Documentation:

This class is responsible for filtering events based on their total pulse height. This class provides functions to configure the pulse height limits of the filter, and to filter events based on the configured limits.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`Filter`**

Public Uses:

`PixelEvent`

Public Interface:

Operations:
`filterEvent()`
`setLimits()`

Private Interface:

Has-A Relationships:

`unsigned` *`lowerLimit`*: This is the lowest event pulse height accepted by the filter.

`unsigned` *`upperLimit`*: This is the largest event pulse height accepted by the filter.

Concurrency: Guarded

Persistence: Transient

37.16.1 filterEvent()

Public member of: **FilterPh**

Return Class: **Boolean**

Arguments:

PixelEvent& *event*

Documentation:

This function filters the pixel event, indicated by *event*, based on its total pulse height. If the event's pulse height is outside the configured limits of the filter, the event is rejected, and this function returns *BoolFalse*. If the event's pulse height is within the filter's limits, the event is accepted, and the function returns *BoolTrue*.

Concurrency: Guarded

37.16.2 setLimits()

Public member of: **FilterPh**

Return Class: **void**

Arguments:

unsigned *lowerBound*
unsigned *range*

Documentation:

This function sets the lower and upper pulse height limits of the filter. Events whose energy is greater than or equal to *lowerBound*, and less than *lowerBound* + *range* are accepted. All others are rejected.

Concurrency: Guarded

37.17 Class PmEvent

Documentation:

This class represents a generic event processor, whose responsibility is to parse records produced by a FEP, and produce data packets and exposure records. This class implements common functions used by all event processor types.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **ProcessMode**

Implementation Uses:

FEPexpRec
FEPexpEndRec
FEPerrorRec
Tf_Record_Event

Public Interface:

Operations:

```
PmEvent()
finishExposure()
processRecord()
setGradeFilter()
setPhFilter()
setWindowFilter()
```

Protected Interface:

Has-A Relationships:

`FilterPh*` *`phFilter`*: This is a pointer to the pulse-height filter for the event processor. It is installed by the client using `setPhFilter`, and is configured directly by the client when setting up the science run.

`FilterWindow*` *`windowFilter`*: This is a pointer to the window list filter for this event processor. It is installed by the client using `setWindowFilter`, and is configured directly by the client when setting up the science run.

`FilterGrade*` *`gradeFilter`*: This is a pointer to the event grade filter for the event processor. It is set by the client using `setGradeFilter`, and is configured directly by the client code.

Operations:

```
digestBiasError()
filterEvent()
incEventCnt()
setupExposureRecord()
```

Private Interface:

Has-A Relationships:

`unsigned` *`parityErrCnt`*: This variable contains the total number of bias map parity errors since the start of the run. This count is set to zero at the start of the run, and is advanced by calls to `digestBiasError`.

`unsigned` *`packedEvents`*: This variable contains the total number of events sent by this exposure. This counter is advanced by calls to `incEventCnt`.

`Tf_Data_Bias_Error` *`biasErrForm`*: This is a bias error telemetry packet form responsible for sending bias map parity errors to telemetry.

Concurrency: Guarded

Persistence: Persistent

37.17.1 PmEvent()

Public member of: **`PmEvent`**

Documentation:

This is the constructor for the **`PmEvent`** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.17.2 digestBiasError()

Protected member of: **`PmEvent`**

Return Class: **`Boolean`**

Arguments:

`FEPerrorRec*` *`record`*

Documentation:

This function handles a bias error record produced by the FEP. *`record`* points to copy of the bias error record. If successful, this function returns *`BoolTrue`*. If the mode is aborted, or an error occurs, the function returns *`BoolFalse`*.

Semantics:

If *`biasErrForm`* does not have a buffer, use `waitForPkt`() to obtain a telemetry packet buffer for the error. Once a buffer is obtained, store the start time, parameter block id and CCD and FEP ids into the buffer (see **`ProcessMode`**::`getTimeBias`() and `getRunIdInfo`()). Append the error row and column to the buffer using *`biasErrForm`*.`append_Bias_Error`s(). If the buffer is full, post it to telemetry using *`biasErrForm`*.`post`().

Concurrency: Guarded

37.17.3 filterEvent()

Protected member of: **PmEvent**

Return Class: **Boolean**

Arguments:

PixelEvent& *event*

Documentation:

This function runs *event* through the process mode's collection of event filters. If the event is accepted by the filter set, the function returns *BoolTrue*. If the event has been rejected, it returns *BoolFalse*. In order for the window event sampling to be based on desired events (i.e. events with the appropriate pulse height, grade and position), the window filter must always be applied AFTER all other filters.

Preconditions:

phFilter, *windowFilter* and *gradeFilter* must be configured and ready to process data.

Semantics:

Apply event to *phFilter*. If accepted, try *gradeFilter*. If still accepted, try *windowFilter*. Return *BoolTrue* if all filters accept the event, and *BoolFalse* if any of the filters reject the event.

Concurrency: Guarded

37.17.4 finishExposure()

Public member of: **PmEvent**

Return Class: **Boolean**

Documentation:

This is an abstract function which must be implemented by each subclass. The function must complete the current exposure, given the mode implementing the function. If successful, the function returns *BoolTrue*. If the run is aborted, or has a fatal error, the function returns *BoolFalse*.

Concurrency: Guarded

37.17.5 incEventCnt()

Protected member of: **PmEvent**

Return Class: **void**

Documentation:

This function increments the telemetered event count for the exposure.

Concurrency: Guarded

37.17.6 processRecord()

Public member of: **PmEvent**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function parses the FEP-produced record contained within *record*. Typically, subclasses of this class overload `processRecord`, handling their own record types and calling this function only if the subclass does not recognize the record type. For example, **PmTeFaint3x3**'s `processRecord` function will handle FEP_EVENT_REC_3x3 record type tags, but pass all other record types to this function. This function recognizes the following record type tags:

FEP_EXPOSURE_REC (**FEPexpRec**)
FEP_EXPOSURE_END_REC (**FEPexpEndRec**)
FEP_ERR_REC (**FEPerrRec**)
TBD: FEP_FID_REC

Semantics:

If the record type is FEP_EXPOSURE_REC, copy the record information into the exposure information instance using *expInfo*.`copyExpStart`(). Zero *packedEvents* and *parityErrors*, and use *phFilter*.`resetCnt`(), *windowFilter*.`resetCnt`(), and *gradeFilter*.`resetCnt`() to reset the filter counters.

If the record type is FEP_EXPOSURE_END_REC, use *expInfo*.`copyExpEnd`() to load exposure end information into the instance. If *biasErrForm* has data, post it to telemetry using *biasErrForm*.`post`(). Then use `finishExposure`() to post the exposure record and perform processing mode specific cleanup actions.

If the record type is FEP_ERR_REC, use `digestBiasError`() to add the bias map parity error the information to the current bias error data telemetry packet.

Concurrency: Guarded

37.17.7 setGradeFilter()

Public member of: **PmEvent**

Return Class: **void**

Arguments:

FilterGrade* *gradefilt*

Documentation:

This function assigns the grade filter, *gradefilt*, to this process instance. If *gradefilt* is 0, no grade filtering is done by this instance.

Concurrency: Guarded

37.17.8 setPhFilter()

Public member of: **PmEvent**

Return Class: **void**

Arguments:

FilterPh* *phfilt*

Documentation:

This function sets the pulse height filter, *phfilt*, for use by this instance. If *phfilt* is 0, then no global pulse height filtering is done by this instance.

Concurrency: Guarded

37.17.9 setWindowFilter()

Public member of: **`PmEvent`**

Return Class: **`void`**

Arguments:

`FilterWindow*` *`windowfilt`*

Documentation:

This function sets the window filter, *`windowfilt`*, to this instance. If *`windowfilt`* is 0, then this instance performs no window filtering.

Concurrency: Guarded

37.17.10 setupExposureRecord()

Protected member of: **PmEvent**

Return Class: **Boolean**

Arguments:

Tf_Record_Event& *form*

Documentation:

This function prepares the exposure record telemetry object, *form*, to accumulate event data.

Concurrency: Guarded

37.18 Class PmHist

Documentation:

This class is responsible for processing histogram data accumulated over a series of exposures by the Front End Processor.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`ProcessMode`**

Public Interface:

Operations:
```
PmHist()
getQuadMode()
processRecord()
setQuadMode()
```

Protected Interface:

Operations: `finishExposure()`

Private Interface:

Has-A Relationships:

`QuadMode` *quadmode*: This is the output register mode being used to produce the histograms.

Concurrency: Guarded

Persistence: Transient

37.18.1 PmHist()

Public member of: **PmHist**

Documentation:

This is the constructor for the **PmHist** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.18.2 finishExposure()

Protected member of: **PmHist**

Return Class: **Boolean**

Documentation:

This function ends the current histogram. Each subclass must implement this function.

Concurrency: Guarded

37.18.3 getQuadMode()

Public member of: **PmHist**

Return Class: **QuadMode**

Documentation:

This function returns the quadrant mode being used to produce the histograms.

Concurrency: Guarded

37.18.4 processRecord()

Public member of: **PmHist**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function is responsible for processing a FEP to BEP ring-buffer record. If successful, the function returns *BoolTrue*. If the run is aborted, the function returns *BoolFalse*. This function recognizes the following record type tags:

FEP_EXPOSURE_REC (**FEPexpRec**)
FEP_EXPOSURE_END_REC (**FEPexpEndRec**)

Semantics:

Switch on the record type. If the record indicates the start of an exposure, use load the information into the exposure record using *expInfo*.copyExpStart(). If the record delimits the end of the histogram exposures, copy the closure information using *expInfo*.copyExpEnd() and then tell the subclass to finish of the histogram, using finishExposure().

Concurrency: Guarded

37.18.5 setQuadMode()

Public member of: **PmHist**

Return Class: **void**

Arguments:

QuadMode *mode*

Documentation:

This function sets the quadrant mode, indicated by *mode*, being used to produce the histograms.

Concurrency: Guarded

37.19 Class PmRaw

Documentation:

This class is responsible for processing raw pixels produced by a single Front End Processor.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **ProcessMode**

Public Uses:

PixelRow

Implementation Uses:

HuffmanMap

Public Interface:

Operations:

```
PmRaw()
getCompression()
getOverclockCnt()
processRecord()
setCompression()
setOverclockCnt()
setWindowFilter()
```

Protected Interface:

Operations:

```
accumulateRawRecord()
filterRow()
finishExposure()
getRawRecord()
packRow()
```

Private Interface:

Has-A Relationships:

`FilterWindow*` *`windowFilter`*: This is a pointer to the window filters to use to clip raw image data.

`HuffmanTable` *`rawHuffman`*: This is the compression table to use when compression raw mode data.

`FEPeventRecRaw` *`accumulator`*: This field is used to accumulate a single FEP raw mode record from a series of 32-bit FEP to BEP ring-buffer records.

`unsigned` *`remaining`*: This is the number of 32-bit words remaining to acquire before the raw mode record is complete.

`unsigned` *`compressCode`*: This is a copy of the compression table code, set by `setCompression`().

`unsigned` *`overclocks`*: This is the total number of overclock pixels in each raw row.

`unsigned*` *`accumPtr`*: This is points to the next location to append to.

Concurrency: Guarded

Persistence: Transient

37.19.1 PmRaw()

Public member of: **PmRaw**

Documentation:

This is the constructor for the **PmRaw** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.19.2 accumulateRawRecord()

Protected member of: **PmRaw**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*
Boolean *start*

Documentation:

This function appends the passed FEP to BEP *record*'s data to the accumulated raw record structure (of type **FEPeventRecRaw**). If *start* is *BoolTrue*, it indicates that this is the first record to accumulate. If *start* is *BoolFalse*, then the passed record is to be appended to an already started raw row. The function returns *BoolTrue* once the raw record is complete, and *BoolFalse* if the record remains incomplete.

Concurrency: Guarded

37.19.3 filterRow()

Protected member of: **PmRaw**

Return Class: **void**

Arguments:

PixelRow& *row*

Documentation:

This function runs the row record, row, through the window list clipping filter, *windowFilter*.

Concurrency: Guarded

37.19.4 finishExposure()

Protected member of: **PmRaw**

Return Class: **Boolean**

Documentation:

This function completes the current exposure. The function returns *BoolTrue* if successful, and *BoolFalse* if the run has been aborted. All subclasses of this class must implement this function.

Concurrency: Guarded

37.19.5 getCompression()

Public member of: **PmRaw**

Return Class: **unsigned**

Documentation:

This function returns the compression table code used to pack data.

Concurrency: Guarded

37.19.6 getOverclockCnt()

Public member of: **PmRaw**

Return Class: **unsigned**

Documentation:

This function returns the total number of overclock pixels in each row.

Concurrency: Guarded

37.19.7 getRawRecord()

Protected member of: **PmRaw**

Return Class: **FEPeventRecRaw***

Documentation:

This function returns a pointer to the completed raw mode record.

Concurrency: Guarded

37.19.8 packRow()

Protected member of: **PmRaw**

Return Class: **Boolean**

Arguments:

const PixelRow& *row*
unsigned*& *dstptr*
unsigned& *dstlen*
Boolean& *partial*
unsigned& *pixels*

Documentation:

This function uses the Huffman compression instance, rawHuffman, to pack the row's data contained in *row*, into the destination buffer pointed to by *dstptr*. The destination buffer contains at least *dstlen* 32-bit words. Upon returning, *dstptr* points to the next location to pack to, and *dstlen* contains the number of words remaining in the destination buffer. If *partial* contains *BoolTrue*, then the last word in the buffer was partially filled, and is not reflected in *dstptr* and *dstlen*. *pixels* will contain the count of pixels packed into the output buffer. If the destination buffer spilled, the write is aborted, and the function returns *BoolFalse*. If the write is complete, then the function returns *BoolTrue*.

Preconditions:

This function assumes that the *row* has been filtered.

Concurrency: Guarded

37.19.9 processRecord()

Public member of: **PmRaw**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a single FEP to BEP data record, *record*. It returns *BoolTrue* if the run is to be continued, and *BoolFalse* if the run has been aborted. This function recognizes the following record type tags:
FEP_EXPOSURE_REC (**FEPexpRec**)
FEP_EXPOSURE_END_REC (**FEPexpEndRec**)

Concurrency: Guarded

37.19.10 setCompression()

Public member of: **PmRaw**

Return Class: **void**

Arguments:

unsigned *code*

Documentation:

This function sets the compression table, indicated by *code*, to use when packing raw mode data.

Concurrency: Guarded

37.19.11 setOverclockCnt()

Public member of: **PmRaw**

Return Class: **void**

Arguments:

unsigned *pixels*

Documentation:

This function sets the total number of overclock pixels in each raw row to the value contained in *pixels*.

Concurrency: Guarded

37.19.12 setWindowFilter()

Public member of: **PmRaw**

Return Class: **void**

Arguments:

FilterWindow* *windows*

Documentation:

This function configures the raw data process to use the window filters pointed to by *windows*.

Concurrency: Guarded

37.20 Class PmTeHist

Documentation:

This class is responsible for processing Timed Exposure Histograms.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **PmHist**

Implementation Uses:

Tf_Record_Te_Histogram
FEPeventRecHist

Public Interface:

Operations:
```
PmTeHist()
processRecord()
```

Protected Interface:

Operations:
```
finishExposure()
finishQuadrant()
isEndOfHistogram()
sendBins()
setupDataPkt()
```

Private Interface:

Has-A Relationships:

`Tf_Data_Te_Hist` *`histForm`*: This is the telemetry packet builder for Timed Exposure histograms.

`QuadCode` *`curQuadrant`*: This identifies which quadrant's histogram is currently being processed.

`unsigned` *`curBin`*: This identifies the current bin number of the histogram being processed.

`Boolean` *`sending`*: This indicates that the mode is acquiring and packing telemetry bins. *`BoolTrue`* indicates that a histogram is being processed, and *`BoolFalse`* indicates that no histograms are being processed.

`unsigned*` *`packPtr`*: This points to the next telemetry buffer location to store histogram bins.

`unsigned` *`packCount`*: This specifies the number of bins remaining to store into the current packet.

`unsigned` *`packWritten`*: This is the total number of bins written into the current buffer.

`PhHistogram` *`histogram`*: This is a histogram instance, used to maintain the header information.

`unsigned` *`packetNum`*: This is the current packet number built for the current histogram.

Concurrency: Guarded

Persistence: Persistent

37.20.1 PmTeHist()

Public member of: **PmTeHist**

Documentation:

This is the constructor for the **PmTeHist** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.20.2 finishExposure()

Protected member of: **PmTeHist**

Return Class: **Boolean**

Documentation:

This function completes the current histogram and issues a histogram report. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

NOTE: Since closure of Timed Exposure histograms is handled as the quadrant histograms are read from the FEP, this function is acting as a hook, and currently only returns *BoolTrue*.

Concurrency: Guarded

37.20.3 finishQuadrant()

Protected member of: **PmTeHist**

Return Class: **Boolean**

Documentation:

This function flushes the last data packet of the current quadrant, and forms and issues the histogram record for the quadrant. It then advances *curQuadrant* to the next quadrant to process. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

Detect if histForm has an un-sent buffer using *histForm*.hasBuffer(). If so, set the final word count and post it to telemetry. Declare a local record form and get a telemetry packet buffer using waitForPkt(). If the wait is aborted, return *BoolFalse*. If a buffer is obtained, then set each field in the packet buffer and post the buffer. Then determine the next quadrant expected from the FEP, update *curQuadrant*, zero *curBin* and return *BoolTrue*.

Concurrency: Guarded

37.20.4 isEndOfHistogram()

Protected member of: **PmTeHist**

Return Class: **Boolean**

Documentation:

This function indicates if the entire histogram has been processed. If it returns *BoolFalse*, there are more histogram bins to process. If it returns *BoolTrue*, then all histogram bins have been handled.

Semantics:

If *curBin* is greater than or equal to the number of bins in a quadrant (4096) and *curQuadrant* corresponds to the last quadrant (node C if in AC mode, and node D in all other modes), then the histogram is complete, otherwise there are more bins to read.

Concurrency: Guarded

37.20.5 processRecord()

Public member of: **PmTeHist**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes the FEP to BEP ring-buffer record, *record*. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*. This function consumes records of type **FEPeventRecHist**, and forwards all others to its parent class, **PmHist**::processRecord().

Semantics:

If already in the process of sending a histogram (*sending* == *BoolTrue*), pass the entire record to sendBins(). If not yet sending a histogram, and the record type indicates the first record of the histogram, copy the header into *histogram*, zero *curBin*, initialize *curQuadrant* to the first node in use, and pass the initial bin values to sendBins(). If not sending a histogram, and the record is not the start of a histogram, let the parent handle it by passing the record to **PmHist**::processRecord().

Concurrency: Guarded

37.20.6 sendBins()

Protected member of: **`PmTeHist`**

Return Class: **`Boolean`**

Arguments:

`const unsigned*` *`binptr`*
`unsigned` *`bincount`*

Documentation:

This function packs and telemeters the histogram bins pointed to by *`binptr`*. The number of bins is specified by *`bincount`*. If successful, the function returns *`BoolTrue`*. If the run is aborted, return *`BoolFalse`*.

Semantics:

If the histogram is complete (`isEndOfHistogram`() == *`BoolTrue`*), set *`sending`* to *`BoolFalse`* and return. If not, enter a loop which terminates when the entire input buffer has been processed, or when the histogram is complete. On each iteration, use `setupDataPkt`() to ensure that the data packet has a telemetry packet buffer. Then copy the bin values into the telemetry packet. If the packet fills, set the number of bin values written, and post the packet to telemetry. If the packet completed a quadrant's histogram (*`curBin`* >= 4096), call `finishQuadrant`() to issue a report for the quadrant's histogram.

Concurrency: Guarded

37.20.7 setupDataPkt()

Protected member of: **PmTeHist**

Return Class: **Boolean**

Documentation:

This function sets up the data packet, *histForm*. If successful, the function returns *BoolTrue*. If the run is aborted, the function returns *BoolFalse*.

Semantics:

Use *histForm*.hasBuffer() to determine if *histForm* has a buffer. If not, use waitForPkt() to allocate a buffer to the form. If the run is aborted while waiting, return *BoolFalse*. Once a buffer is obtained, set the header information in the packet buffer. Then set *packPtr* to the start of the bin data area within the packet, and *packCount* to the number of words available in the buffer. Zero *packWritten*. If *packCount* holds more values than are remaining for the current quadrant (*curBin* + *packCount* > bins/quadrant (i.e. 4096)), truncate *packCount* to match the number of remaining bins.

Concurrency: Guarded

37.21 Class PmTeRaw

Documentation:

This class processes raw Timed Exposure Mode data.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **PmRaw**

Implementation Uses:

FEPeventRecRaw
Tf_Record_Te_Raw

Public Interface:

Operations:
```
PmTeRaw()
processRecord()
```

Protected Interface:

Operations:
```
finishExposure()
digestRawRecord()
setupDataPkt()
```

Private Interface:

Has-A Relationships:

`Tf_Data_Te_Raw` *`rawForm`*: This is the Timed Exposure Raw Mode data packet.

`unsigned` *`rowsPacked`*: This indicates the number of raw rows packed into the data packet.

`Boolean` *`accumulating`*: This indicates whether or not the mode is accumulating FEP to BEP records into a single record. *`BoolFalse`* indicates that a raw data record is not being accumulated. *`BoolTrue`* indicates that the data record is being accumulated.

`unsigned*` *`packPtr`*: This is the current position in the data packet where the next row is to be packed.

`unsigned` *`packLen`*: This is the space remaining in the telemetry packet buffer.

`unsigned` *`packWritten`*: This is the total number of words currently written into the telemetry packet buffer.

`Boolean` *`packPartial`*: This indicates that the last write to the telemetry packet buffer left a partially written word at the end of the packet. If this is *`BoolTrue`* at the point at which the buffer is to be sent, *`packWritten`* should be advanced by 1 to include the last word in the buffer.

`unsigned` *`packPixels`*: This is the total number of pixels packed into telemetry for the current exposure.

`unsigned` *`packetNum`*: This is the current number of data packets sent so far in the exposure.

Concurrency: Guarded

Persistence: Transient

37.21.1 PmTeRaw()

Public member of: **PmTeRaw**

Documentation:

This is the constructor for the **PmTeRaw** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.21.2 digestRawRecord()

Protected member of: **PmTeRaw**

Return Class: **Boolean**

Arguments:

FEPeventRecRaw* *record*

Documentation:

This function processes the raw mode record, *record*. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

Declare a **PixelRow** instance, *row*, invoke setup() and attachData() to associate the row instance with the passed *record*. Call **PmRaw**::filterRow() to apply the clipping windows. Then use setupDataPkt() to ensure that *rawForm* has a telemetry packet buffer. Call **PmRaw**::packRow() to pack the row into the telemetry packet. If the row doesn't fit into the buffer, post the current buffer to telemetry. Then call setupDataPkt() to assign a new telemetry packet buffer, and call packRow() again to load the row into the new buffer. Once the row has been packed, advance *packPixels* and *packWritten* by the number of pixels copied and the number of words written into the buffer, respectively.

Concurrency: Guarded

37.21.3 finishExposure()

Protected member of: **PmTeRaw**

Return Class: **Boolean**

Documentation:

This function ends the current exposure for Timed Exposure Raw mode. If successful, this function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

If rawForm has an un-sent data buffer, post the remaining data packet using *rawForm*.post(). Declare a raw mode record and obtain a telemetry packet buffer using waitForPkt(). Set the various fields in the packet buffer, and post it to telemetry. If the run is aborted while waiting for a packet buffer, return *BoolFalse*. If not, return *BoolTrue*.

Concurrency: Guarded

37.21.4 processRecord()

Public member of: **PmTeRaw**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes the FEP to BEP record contained in *record*. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

If in the process of accumulating a raw mode record from the FEP (*accumulating* == *BoolTrue*), call accumulateRawRecord(). If the record completes, set accumulating to *BoolFalse* and call digestRawRecord() to process the completed row. If not in the process of accumulating a record, and the record type indicates the start of a row record, call accumulateRawRecord() to start the accumulation. If not processing a raw mode record, let the parent handle the record using **PmRaw**::processRecord().

Concurrency: Guarded

37.21.5 setupDataPkt()

Protected member of: **PmTeRaw**

Return Class: **Boolean**

Arguments:

PixelRow& *row*

Documentation:

This function acquires a telemetry packet buffer for the data packet, *rawForm*, and initializes the header fields in the data packet. *row* is the first row to attempt to be packed into the telemetry packet buffer.

Semantics:

If *rawForm* does not have a buffer, use waitForPkt() to obtain a buffer for the form. If the run is aborted while waiting, return *BoolFalse*. Once a buffer is obtained set the fields in the packet header, using *row*.getRange() to get the row number. Set *packPtr* to the data area of the packet buffer and *packLen* to the number of data words available. Zero *packWritten*.

Concurrency: Guarded

37.22 Class PmTeFaint3x3

Documentation:

This class is responsible for processing 3x3 Timed Exposure Events, and producing Faint Mode telemetry data. This class provides functions to Front End Processor event data records, to pack and post to telemetry Faint Mode 3x3 events, and to produce Faint Mode exposure records.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`PmEvent`**

Implementation Uses:

`Tf_Record_Te_Faint`
`FEPeventRec3x3`
`Pixel3x3`

Public Interface:

Operations:
`PmTeFaint3x3()`
`processRecord()`

Protected Interface:

Operations:
`finishExposure()`
`sendEvent()`

Private Interface:

Has-A Relationships:

`Tf_Data_Te_Faint` *`dataForm`*: This is the telemetry builder used by the mode to accumulate and format 3x3 event pixel data.

`unsigned` *`packetNum`*: This is the data packet number for the current exposure.

Concurrency: Guarded

Persistence: Persistent

37.22.1 PmTeFaint3x3()

Public member of: **PmTeFaint3x3**

Documentation:

This is the constructor for the **PmTeFaint3x3** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.22.2 finishExposure()

Protected member of: **PmTeFaint3x3**

Return Class: **Boolean**

Documentation:

This function completes the current exposure, forming and sending an exposure record. It returns *BoolTrue* on success, and *BoolFalse* on error or abort request.

Semantics:

Check *dataForm*.hasBuffer() and if the packet buffer has data, post it. Declare an exposure record form and call **PmEvent**::setupExposureRecord() to obtain its telemetry buffer and store common exposure record information into the buffer. The call *form*.post() to queue the packet buffer to telemetry.

Concurrency: Guarded

37.22.3 processRecord()

Public member of: **PmTeFaint3x3**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a record produced by the FEP, contained within *record*. This function explicitly handles the following record tag types:

FEP_EVENT_REC_3x3 (**FEPeventRec3x3**)

It uses its parent's member function, **PmEvent**::processRecord() to handle all other record types. This function returns *BoolTrue* if the record was processed, and *BoolFalse* on an error or abort request.

Semantics:

Check the first record tag for FEP_EVENT_REC_3x3. If it is not this type, call **PmEvent**::processRecord() to attempt to handle the record. If the record type matches, declare **Pixel3x3** instance, and pass it *expInfo*, and a pointer to the record to **Pixel3x3**::attachData(). *expInfo* supplies the new event with the configured split threshold levels and exposure overclock levels, and coverts the event's pixel position to absolute CCD coordinates. Then call **PmEvent**::filterEvent(). If filterEvent() accepts the event, call sendEvent() to pack it into the data buffer.

Concurrency: Guarded

37.22.4 sendEvent()

Protected member of: **PmTeFaint3x3**

Return Class: **Boolean**

Arguments:

Pixel3x3& *event*

Documentation:

This function processes an accepted event, indicated by *event*, produced by the FEP, storing the event into *dataForm*'s telemetry buffer, and posting *dataForm*'s buffer when and if it becomes full. This function returns *BoolTrue* if the event was processed, and *BoolFalse* on an error or abort request.

Semantics:

If *dataForm* does not already have a packet buffer, call waitForPkt() to obtain a telemetry packet, and set the header information in the packet. If the run is aborted while waiting for a packet, return *BoolFalse*. Use the passed event to get the CCD position of the event, and the event's raw pulse height values and append them to the current telemetry packet. Increment the mode's event counter using **PixelEvent**::incEventCnt(). If the telemetry buffer is full, post it to telemetry.

Concurrency: Guarded

37.23 Class PmTeFaintBias3x3

Documentation:

This class is responsible for processing 3x3 Timed Exposure Events, and producing Faint-with-Bias Mode telemetry data. This class provides functions to Front End Processor event data records, to pack and post to telemetry Faint Mode 3x3 events, and to produce Faint-with-Bias Mode exposure records.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`PmEvent`**

Implementation Uses:

`Tf_Record_Te_Faint_Bias`
`FEPeventRec3x3`
`Pixel3x3`

Public Interface:

Operations: `PmTeFaintBias3x3()`
`processRecord()`

Protected Interface:

Operations: `finishExposure()`
`sendEvent()`

Private Interface:

Has-A Relationships:

`Tf_Data_Te_Faint_Bias` *`dataForm`*: This is the telemetry builder used by the mode to accumulate and format 3x3 event pixel and bias data.

`unsigned` *`packetNum`*: This is the data packet number for the current exposure.

Concurrency: Guarded

Persistence: Persistent

37.23.1 PmTeFaintBias3x3()

Public member of: **PmTeFaintBias3x3**

Documentation:

This is the constructor for the **PmTeFaintBias3x3** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.23.2 finishExposure()

Protected member of: **PmTeFaintBias3x3**

Return Class: **Boolean**

Documentation:

This function completes the current exposure, forming and sending an exposure record. It returns *BoolTrue* on success, and *BoolFalse* on error or abort request.

Semantics:

Check *dataForm*.hasBuffer() and if the packet buffer has data, post it. Declare an exposure record form and call **PmEvent**::setupExposureRecord() to obtain its telemetry buffer and store common exposure record information into the buffer. Then add the bias offsets to the exposure record post the packet buffer to telemetry.

Concurrency: Guarded

37.23.3 processRecord()

Public member of: **PmTeFaintBias3x3**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a record produced by the FEP, contained within *record*. This function explicitly handles the following record tag types:

FEP_EVENT_REC_3x3 (**FEPeventRec3x3**)

It uses its parent's member function, **PmEvent**::processRecord() to handle all other record types. This function returns *BoolTrue* if the record was processed, and *BoolFalse* on an error or abort request.

Semantics:

Check the first record tag for FEP_EVENT_REC_3x3. If it is not this type, call **PmEvent**::processRecord() to attempt to handle the record. If the record type matches, declare **Pixel3x3** instance, and pass it *expInfo*, and a pointer to the record to **Pixel3x3**::attachData(). *expInfo* supplies the new event with the configured split threshold levels and exposure overclock levels, and converts the event's pixel position to absolute CCD coordinates. Then call **PmEvent**::filterEvent(). If filterEvent() accepts the event, call sendEvent() to pack it into the data buffer.

Concurrency: Guarded

37.23.4 sendEvent()

Protected member of: **PmTeFaintBias3x3**

Return Class: **Boolean**

Arguments:

Pixel3x3& *event*

Documentation:

This function processes an accepted event, indicated by *event*, produced by the FEP, storing the event into *dataForm*'s telemetry buffer, and posting *dataForm*'s buffer when and if it becomes full. This function returns *BoolTrue* if the event was processed, and *BoolFalse* on an error or abort request.

Semantics:

If *dataForm* does not already have a packet buffer, call `waitForPkt`() to obtain a telemetry packet, and set the header information in the packet. If the run is aborted while waiting for a packet, return *BoolFalse*. Use the passed event to get the CCD position of the event, and the event's raw pulse height values, and bias values and append them to the current telemetry packet. Increment the mode's event counter using **PixelEvent**::`incEventCnt`(). If the telemetry buffer is full, post it to telemetry.

Concurrency: Guarded

37.24 Class PmTeGraded

Documentation:

This class is responsible for processing 3x3 Timed Exposure Events, and producing Graded Mode telemetry data. This class provides functions to Front End Processor event data records, to pack and post to telemetry Graded Mode events, and to produce Faint Mode exposure records (Graded Mode uses the same exposure records as Faint Mode).

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`PmEvent`**

Implementation Uses:

`Tf_Record_Te_Faint`
`FEPeventRec3x3`
`Pixel3x3`

Public Interface:

Operations:
`PmTeGraded()`
`processRecord()`

Protected Interface:

Operations:
`finishExposure()`
`sendEvent()`

Private Interface:

Has-A Relationships:

`Tf_Data_Te_Graded` *`dataForm`*: This is the telemetry builder used by the mode to accumulate and format graded event pixel data.

`unsigned` *`packetNum`*: This is the data packet number for the current exposure.

Concurrency: Guarded

Persistence: Persistent

37.24.1 PmTeGraded()

Public member of: **PmTeGraded**

Documentation:

This is the constructor for the **PmTeGraded** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.24.2 finishExposure()

Protected member of: **PmTeGraded**

Return Class: **Boolean**

Documentation:

This function completes the current exposure, forming and sending an exposure record. It returns *BoolTrue* on success, and *BoolFalse* on error or abort request.

Semantics:

Check *dataForm*.hasBuffer() and if the packet buffer has data, post it. Declare an exposure record form and call **PmEvent**::setupExposureRecord() to obtain its telemetry buffer and store common exposure record information into the buffer. Then call *form*.post() to queue the packet buffer to telemetry.

Concurrency: Guarded

37.24.3 processRecord()

Public member of: **PmTeGraded**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a record produced by the FEP, contained within *record*. This function explicitly handles the following record tag types:

FEP_EVENT_REC_3x3 (**FEPeventRec3x3**)

It uses its parent's member function, **PmEvent**::processRecord() to handle all other record types. This function returns *BoolTrue* if the record was processed, and *BoolFalse* on an error or abort request.

Semantics:

Check the first record tag for FEP_EVENT_REC_3x3. If it is not this type, call **PmEvent**::processRecord() to attempt to handle the record. If the record type matches, declare **Pixel3x3** instance, and pass it *expInfo*, and a pointer to the record to **Pixel3x3**::attachData(). *expInfo* supplies the new event with the configured split threshold levels and exposure overclock levels, and coverts the event's pixel position to absolute CCD coordinates. Then call **PmEvent**::filterEvent(). If filterEvent() accepts the event, call sendEvent() to pack it into the data buffer.

Concurrency: Guarded

37.24.4 sendEvent()

Protected member of: **PmTeGraded**

Return Class: **Boolean**

Arguments:

Pixel3x3& *event*

Documentation:

This function processes an accepted event, indicated by *event*, produced by the FEP, storing the event into *dataForm*'s telemetry buffer, and posting *dataForm*'s buffer when and if it becomes full. This function returns *BoolTrue* if the event was processed, and *BoolFalse* on an error or abort request.

Semantics:

If *dataForm* does not already have a packet buffer, call waitForPkt() to obtain a telemetry packet, and set the header information in the packet. If the run is aborted while waiting for a packet, return *BoolFalse*. Use the passed event to get the CCD position of the event, the events amplitude and grade code, and to sum the corner pixel pulse heights. Append the acquired information to the current telemetry packet. Increment the mode's event counter using **PixelEvent**::incEventCnt(). If the telemetry buffer is full, post it to telemetry.

Concurrency: Guarded

37.25 Class PmCcRaw

Documentation:

This class processes raw Continuous Clocking Mode data.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **PmRaw**

Implementation Uses:

FEPeventRecRaw
Tf_Record_Cc_Raw

Public Interface:

Operations:
```
PmCcRaw()
processRecord()
```

Protected Interface:

Operations:
```
finishExposure()
digestRawRecord()
setupDataPkt()
```

Private Interface:

Has-A Relationships:

`Tf_Data_Cc_Raw` *`rawForm`*: This is the Continuous Clocking Raw Mode data packet.

`unsigned` *`rowsPacked`*: This indicates the number of raw rows packed into the data packet.

`Boolean` *`accumulating`*: This indicates whether or not the mode is accumulating FEP to BEP records into a single record. *`BoolFalse`* indicates that a raw data record is not being accumulated. *`BoolTrue`* indicates that the data record is being accumulated.

`unsigned*` *`packPtr`*: This is the current position in the data packet where the next row is to be packed.

`unsigned` *`packLen`*: This is the space remaining in the telemetry packet buffer.

`unsigned` *`packWritten`*: This is the total number of words currently written into the telemetry packet buffer.

`Boolean` *`packPartial`*: This indicates that the last write to the telemetry packet buffer left a partially written word at the end of the packet. If this is *`BoolTrue`* at the point at which the buffer is to be sent, *`packWritten`* should be advanced by 1 to include the last word in the buffer.

`unsigned` *`packPixels`*: This is the total number of pixels packed into telemetry for the current exposure.

`unsigned` *`packetNum`*: This is the current number of data packets sent so far in the exposure.

Concurrency: Guarded

Persistence: Transient

37.25.1 PmCcRaw()

Public member of: **PmCcRaw**

Documentation:

This is the constructor for the **PmCcRaw** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.25.2 digestRawRecord()

Protected member of: **PmCcRaw**

Return Class: **Boolean**

Arguments:

FEPeventRecRaw* *record*

Documentation:

This function processes the raw mode record, *record*. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

Declare a **PixelRow** instance, *row*, invoke setup() and attachData() to associate the row instance with the passed *record*. Call **PmRaw**::filterRow() to apply the clipping windows. Then use setupDataPkt() to ensure that *rawForm* has a telemetry packet buffer. Call **PmRaw**::packRow() to pack the row into the telemetry packet. If the row doesn't fit into the buffer, post the current buffer to telemetry. Then call setupDataPkt() to assign a new telemetry packet buffer, and call packRow() again to load the row into the new buffer. Once the row has been packed, advance *packPixels* and *packWritten* by the number of pixels copied and the number of words written into the buffer, respectively.

Concurrency: Guarded

37.25.3 finishExposure()

Protected member of: **PmCcRaw**

Return Class: **Boolean**

Documentation:

This function ends the current exposure for Continuous Clocking Raw mode. If successful, this function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

If *rawForm* has an un-sent data buffer, post the remaining data packet using *rawForm*.post(). Declare a raw mode record and obtain a telemetry packet buffer using waitForPkt(). Set the various fields in the packet buffer, and post it to telemetry. If the run is aborted while waiting for a packet buffer, return *BoolFalse*. If not, return *BoolTrue*.

Concurrency: Guarded

37.25.4 processRecord()

Public member of: **PmCcRaw**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes the FEP to BEP record contained in *record*. If successful, the function returns *BoolTrue*. If the run is aborted, it returns *BoolFalse*.

Semantics:

If in the process of accumulating a raw mode record from the FEP (*accumulating* == *BoolTrue*), call accumulateRawRecord(). If the record completes, set accumulating to *BoolFalse* and call digestRawRecord() to process the completed row. If not in the process of accumulating a record, and the record type indicates the start of a row record, call accumulateRawRecord() to start the accumulation. If not processing a raw mode record, let the parent handle the record using **PmRaw**::processRecord().

Concurrency: Guarded

37.25.5 setupDataPkt()

Protected member of: **PmCcRaw**

Return Class: **Boolean**

Arguments:

PixelRow& *row*

Documentation:

This function acquires a telemetry packet buffer for the data packet, *rawForm*, and initializes the header fields in the data packet. *row* is the first row to attempt to be packed into the telemetry packet buffer.

Semantics:

If *rawForm* does not have a buffer, use waitForPkt() to obtain a buffer for the form. If the run is aborted while waiting, return *BoolFalse*. Once a buffer is obtained set the fields in the packet header, using *row*.getRange() to get the row number. Set *packPtr* to the data area of the packet buffer and *packLen* to the number of data words available. Zero *packWritten*.

Concurrency: Guarded

37.26 Class PmCcFaint1x3

Documentation:

This class is responsible for processing 1x3 Continuous Clocking Events, and producing Faint Mode telemetry data. This class provides functions to Front End Processor event data records, to pack and post to telemetry Faint Mode 1x3 events, and to produce Faint Mode exposure records.

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`PmEvent`**

Implementation Uses:

`Tf_Record_Cc_Faint`
`FEPeventRec1x3`
`Pixel1x3`

Public Interface:

Operations:
`PmCcFaint1x3()`
`processRecord()`

Protected Interface:

Operations:
`finishExposure()`
`sendEvent()`

Private Interface:

Has-A Relationships:

`Tf_Data_Cc_Faint` *`dataForm`*: This is the telemetry builder used by the mode to accumulate and format 1x3 event pixel data.

`unsigned` *`packetNum`*: This is the data packet number for the current exposure.

Concurrency: Guarded

Persistence: Persistent

37.26.1 PmCcFaint1x3()

Public member of: **PmCcFaint1x3**

Documentation:

This is the constructor for the **PmCcFaint1x3** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.26.2 finishExposure()

Protected member of: **PmCcFaint1x3**

Return Class: **Boolean**

Documentation:

This function completes the current exposure, forming and sending an exposure record. It returns *BoolTrue* on success, and *BoolFalse* on error or abort request.

Semantics:

Check *dataForm*.hasBuffer() and if the packet buffer has data, post it. Declare an exposure record form and call **PmEvent**::setupExposureRecord() to obtain its telemetry buffer and store common exposure record information into the buffer. Then call *form*.post() to queue the packet buffer to telemetry.

Concurrency: Guarded

37.26.3 processRecord()

Public member of: **PmCcFaint1x3**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a record produced by the FEP, contained within *record*. This function explicitly handles the following record tag types:

FEP_EVENT_REC_1x3 (**FEPeventRec1x3**)

It uses its parent's member function, **PmEvent**::processRecord() to handle all other record types. This function returns *BoolTrue* if the record was processed, and *BoolFalse* on an error or abort request.

Semantics:

Check the first record tag for FEP_EVENT_REC_1x3. If it is not this type, call **PmEvent**::processRecord() to attempt to handle the record. If the record type matches, declare **Pixel1x3** instance, and pass it *expInfo*, and a pointer to the record to **Pixel1x3**::attachData(). *expInfo* supplies the new event with the configured split threshold levels and exposure overclock levels, and converts the event's pixel position to absolute CCD coordinates. Then call **PmEvent**::filterEvent(). If filterEvent() accepts the event, call sendEvent() to pack it into the data buffer.

Concurrency: Guarded

37.26.4 sendEvent()

Protected member of: **PmCcFaint1x3**

Return Class: **Boolean**

Arguments:

Pixel1x3& *event*

Documentation:

This function processes an accepted event, indicated by *event*, produced by the FEP, storing the event into *dataForm*'s telemetry buffer, and posting *dataForm*'s buffer when and if it becomes full. This function returns *BoolTrue* if the event was processed, and *BoolFalse* on an error or abort request.

Semantics:

If *dataForm* does not already have a packet buffer, call waitForPkt() to obtain a telemetry packet, and set the header information in the packet. If the run is aborted while waiting for a packet, return *BoolFalse*. Use the passed event to get the CCD position of the event, and the event's raw pulse height values and append them to the current telemetry packet. Increment the mode's event counter using **PixelEvent**::incEventCnt(). If the telemetry buffer is full, post it to telemetry.

Concurrency: Guarded

37.27 Class PmCcGraded

Documentation:

This class is responsible for processing 1x3 Continuous Clocking Events, and producing Graded Mode telemetry data. This class provides functions to Front End Processor event data records, to pack and post to telemetry Graded Mode events, and to produce Faint Mode exposure records (Graded mode and Faint mode use the same exposure record type).

Export Control: Public

Cardinality: 6

Hierarchy:

Superclasses: **`PmEvent`**

Implementation Uses:

`Tf_Record_Cc_Faint`
`FEPeventRec1x3`
`Pixel1x3`

Public Interface:

Operations:
`PmCcGraded()`
`processRecord()`

Protected Interface:

Operations:
`finishExposure()`
`sendEvent()`

Private Interface:

Has-A Relationships:

`Tf_Data_Cc_Graded` *`dataForm`*: This is the telemetry builder used by the mode to accumulate and format 1x3 event pixel data.

`unsigned` *`packetNum`*: This is the data packet number for the current exposure.

Concurrency: Guarded

Persistence: Persistent

37.27.1 PmCcGraded()

Public member of: **PmCcGraded**

Documentation:

This is the constructor for the **PmCcGraded** class. This function sets the instance variables of the class to default values.

Concurrency: Guarded

37.27.2 finishExposure()

Protected member of: **PmCcGraded**

Return Class: **Boolean**

Documentation:

This function completes the current exposure, forming and sending an exposure record. It returns *BoolTrue* on success, and *BoolFalse* on error or abort request.

Semantics:

Check *dataForm*.hasBuffer() and if the packet buffer has data, post it. Declare an exposure record form and call **PmEvent**::setupExposureRecord() to obtain its telemetry buffer and store common exposure record information into the buffer. Then call *form*.post() to queue the packet buffer to telemetry.

Concurrency: Guarded

37.27.3 processRecord()

Public member of: **PmCcGraded**

Return Class: **Boolean**

Arguments:

const RINGREC& *record*

Documentation:

This function processes a record produced by the FEP, contained within *record*. This function explicitly handles the following record tag types:

FEP_EVENT_REC_1x3 (**FEPeventRec1x3**)

It uses its parent's member function, **PmEvent**::processRecord() to handle all other record types. This function returns *BoolTrue* if the record was processed, and *BoolFalse* on an error or abort request.

Semantics:

Check the first record tag for FEP_EVENT_REC_1x3. If it is not this type, call **PmEvent**::processRecord() to attempt to handle the record. If the record type matches, declare **Pixel1x3** instance, and pass it *expInfo*, and a pointer to the record to **Pixel1x3**::attachData(). *expInfo* supplies the new event with the configured split threshold levels and exposure overclock levels, and converts the event's pixel position to absolute CCD coordinates. Then call **PmEvent**::filterEvent(). If filterEvent() accepts the event, call sendEvent() to pack it into the data buffer.

Concurrency: Guarded

37.27.4 sendEvent()

Protected member of: **PmCcGraded**

Return Class: **Boolean**

Arguments:

Pixel1x3& *event*

Documentation:

This function processes an accepted event, indicated by *event*, produced by the FEP, storing the event into *dataForm*'s telemetry buffer, and posting *dataForm*'s buffer when and if it becomes full. This function returns *BoolTrue* if the event was processed, and *BoolFalse* on an error or abort request.

Semantics:

If *dataForm* does not already have a packet buffer, call `waitForPkt`() to obtain a telemetry packet, and set the header information in the packet. If the run is aborted while waiting for a packet, return *BoolFalse*. Use the passed event to get the CCD position of the event, and the event's amplitude and grade and append them to the current telemetry packet. Increment the mode's event counter using **PixelEvent**::`incEventCnt`(). If the telemetry buffer is full, post it to telemetry.

Concurrency: Guarded

38.0 Bias Thief Class (36-53239 01)

38.1 Purpose

The purpose of the Bias Thief is to copy the contents the Front End Processor (FEP) pixel bias map values to telemetry during science processing. The Bias Thief copies the values directly from the FEP's bias map in BEP-FEP shared-memory without directly interacting with the software running on the FEP, hence the name "thief."

As is for other telemetry producers, its telemetry utilization is bounded by the number of telemetry packet buffers allocated to it by the system during startup. By convention, the Science Manager (see Section 33.0) is allocated the bulk of the telemetry buffers, hence the Bias Thief tends to trickle the maps to telemetry when telemetry is saturated with science data.

38.2 Uses

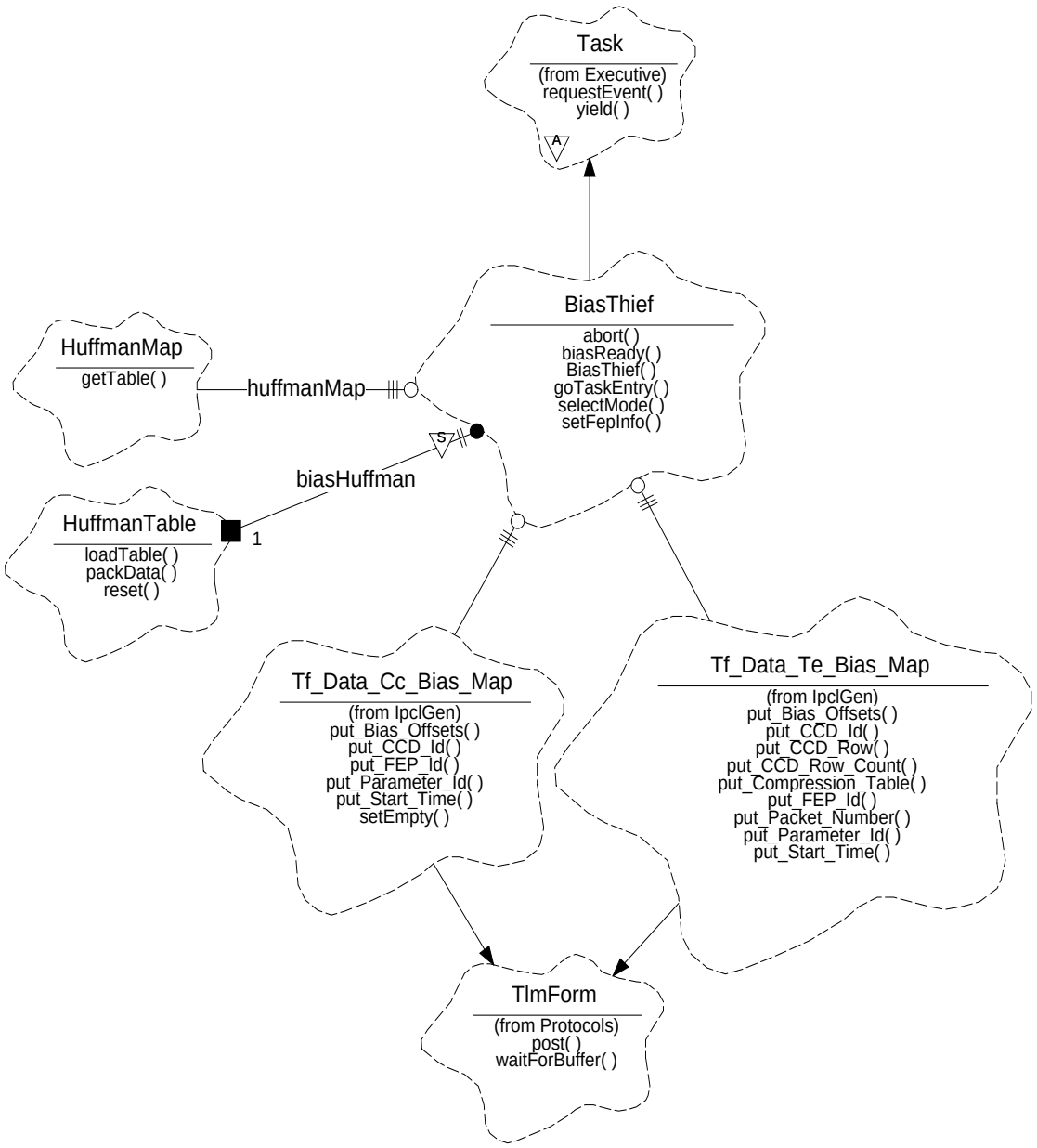
The following lists the use of the Bias Thief class:

Use 1:: Select which type of bias maps are to be sent
Use 2:: Specify the bias map parameters for each Front End Processor
Use 3:: Start transmission of the pixel bias maps
Use 4:: Abort transmission of the pixel bias maps

38.3 Organization

Figure 184 illustrates the class relationships used by the **`BiasThief`** class.

FIGURE 184. Bias Thief Class Relationships



`BiasThief`- This class is a subclass of ***`Executive`***::**`Task`** and is responsible for sending the pixel bias maps to telemetry while science processing is underway. This class provides a function which selects the type of pixel bias map to be sent, Timed Exposure or Continuous Clocking (`selectMode`). It provides a function to load the properties of a given FEP's bias map, such as which CCD was being processed, what the initial overclock values were, etc. (`setFepInfo`). The class provides functions to start the bias operation (`biasReady`) and to abort the transmission of the bias maps (`abort`).

`Task`- This class is supplied by the ***`Executive`*** class category. It represents and controls an active running task. The **`BiasThief`** class inherits from this class, and uses the class's

functions to relinquish control to allow other tasks of the same priority to run (`yield`), and to detect queries from the **`TaskMonitor`** (`requestEvent`).

`TaskMonitor` (not shown)- This class is supplied by the ***`Executive`*** class category, and is responsible for periodically polling each task in the instrument. When polled, the **`BiasThief`** task responds using this classes member function (`respond`).

`HuffmanMap` - This class maintains the collection of Huffman compression tables store in I-cache. It provides functions to map an table index to the address in I-cache corresponding to the selected table (`getTable`).

`HuffmanTable` - This class is responsible for compressing data using a selectable compression table. It provides functions which load a table from I-cache (loadTable), to reset its state-machine to start compressing a set of data (reset), and to compress input data and append the data to a user-supplied output buffer (packData).

`Tf_Data_Cc_Bias_Map` - This class is generated by the IP&CL code-generator, and belongs to the ***`IpclGen`*** class category). It is a subclass of ***`Protocols`***::**`TlmForm`** and is responsible for formatting a Continuous Clocking Bias Map telemetry packet. It provides functions which write the initial overclock values for the current bias map (`put_Bias_Offsets`), write the CCD identifier used to produce the map (`put_CCD_Id`), write the identifier of the FEP which produced the map (`put_FEP_Id`), write the parameter block id used to compute the bias map (`put_Parameter_Id`), and write the ACIS science timestamp, latched at the start of the bias computation (`put_Start_Time`). It also provides a function which resets the contents of the bias map data (`setEmpty`), and provides functions which return the address and length of the bias map data buffer within the packet and set the number of 32-bit words written into the buffer (`get_Data_Address`, `get_Data_Avail`, `set_Data_Written`, not shown).

`Tf_Data_Te_Bias_Map` -This class is generated by the IP&CL code-generator, and belongs to the ***`IpclGen`*** class category). It is a subclass of ***`Protocols`***::**`TlmForm`** and is responsible for formatting a Timed Exposure Bias Map telemetry packet. It provides functions which write the initial overclock values for the current bias map (`put_Bias_Offsets`), write the CCD identifier used to produce the map (`put_CCD_Id`), write the identifier of the FEP which produced the map (`put_FEP_Id`), write the parameter block id used to compute the bias map (`put_Parameter_Id`), and write the ACIS science timestamp, latched at the start of the bias computation (`put_Start_Time`). It provides functions which write the starting CCD row identifier into the packet, sets the number of rows written into the packet buffer (`put_CCD_Row`, `put_CCD_Row_Count`), and writes the compression table identifier used to pack the data (`put_Compression_Table`). It also provides a function which resets the contents of the bias map data (`setEmpty`, not shown), sets the bias data packet number (`put_Packet_Number`) and provides functions which return the address and length of the bias map data buffer within the packet and set the number of 32-bit words written into the buffer (`get_Data_Address`, `get_Data_Avail`, `set_Data_Written`, not shown).

`TlmForm` - This class is provided by the ***`Protocols`*** class category, and is responsible for overall formatting of telemetry packet buffers. It provides functions which wait for and allocate a telemetry packet buffer (`waitForBuffer`), and which post the buffer for transfer to telemetry (`post`).

38.4 Scenarios

38.4.1 Use 1: Select which type of bias maps are to be sent

Prior to packing data, the *client* must select which type of bias maps are to be telemetered, and specify the start time of the bias computation to be telemetered, and parameter block id used to compute the maps by calling *biasThief*.selectMode(). selectMode() then records the information within the *biasThief*, and clears all FEP-specific information within the biasThief.

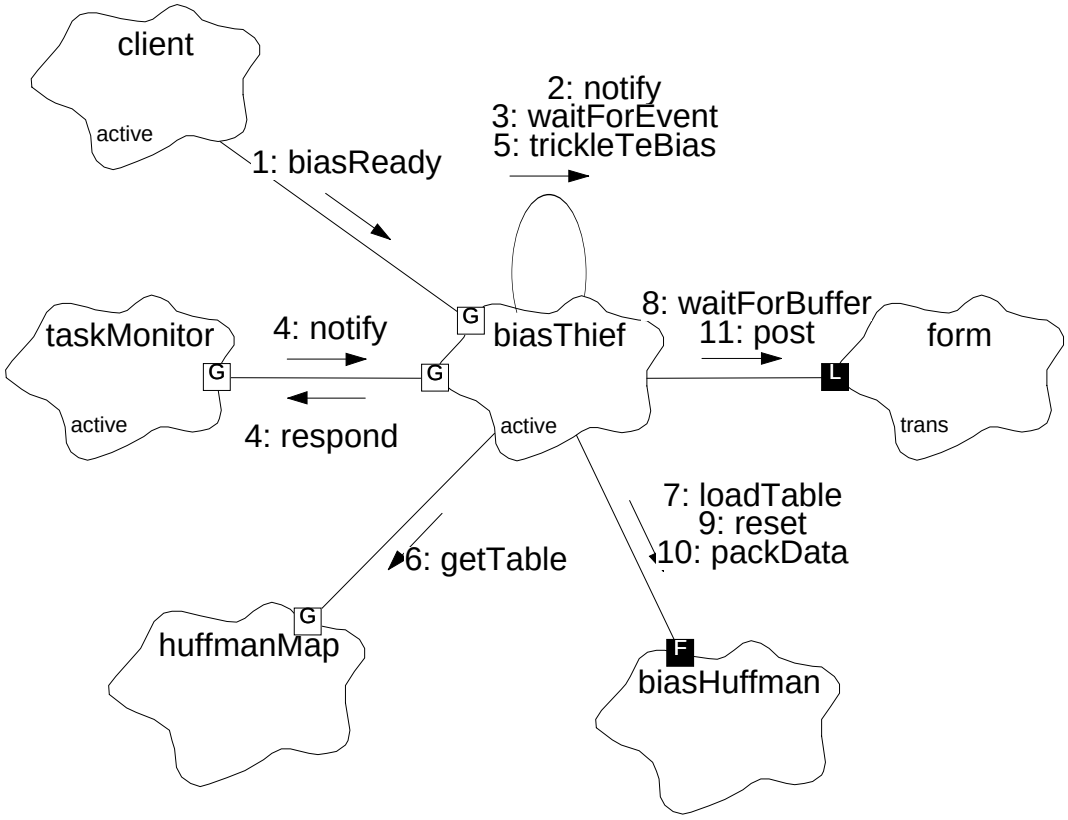
38.4.2 Use 2: Specify the bias map parameters for each Front End Processor

After selecting the bias map mode and prior to packing data, the client must specify various FEP-specific parameters for each bias map to send, using *biasThief*.setFepInfo(). This function selects which FEPs to use, which CCD produced the map, what the bias map base address is (in BEP address space), how many bias pixels there are per row for that FEP, how many rows there are in the map, what the initial overclock values are for the map, and which compression table to use for this map. After calling this function for each configured FEP, the *biasThief* is ready to telemeter maps.

38.4.3 Use 3: Start transmission of the pixel bias maps

Figure 185 illustrates the overall process used to telemetry the pixel bias maps from each configured FEP. Some details have been omitted to simplify the explanation.

FIGURE 185. Trickle Pixel Bias Maps



1. In order to start sending the configured bias maps, the client informs the biasThief that the maps are ready, using the binding function, *`biasThief`*.`biasReady`().
2. *`biasThief`*.`biasReady`() de-asserts the abort flag, and calls `notify`() to inform the task portion that the map is ready.
3. The main task loop, `goTaskEntry`, waits for notification that it should start, using `waitForEvent`().
4. While waiting, if it receives a query from the taskMonitor, it replies using *`taskMonitor`*.`respond`().
5. If it `waitForEvent`() receives a start signal, `goTaskEntry` iterates through each FEP until each of the bias maps have been sent, or until it is aborted. If a particular FEP is configured, `goTaskEntry`() calls `trickleTeBias`() or `trickleCcBias`(), depending on the mode specified by `selectMode`(), to telemeter that FEP's map. For this example, assume that a Timed Exposure bias is to be sent.
6. `trickleTeBias`() gets the address of the desired compression table using *`huffmapMap`*.`getTable`().

7. It passes this address to *biasHuffman*.loadTable() to copy the table from protected I-cache memory into a usable buffer.
8. trickleTeBias() constructs a telemetry format object, form, which is used to format the bias map's telemetry buffer. It obtains a buffer for the form using getBuffer() (not shown), which then calls form.waitForBuffer(). getBuffer() uses the time-out feature of form.waitForBuffer() to occasionally respond to taskMonitor queries and detect abort requests.
9. If a buffer is obtained without aborting the operation, trickleTeBias() initiates a new compression operation using *biasHuffman*.reset().
10. It then compresses a set rows of pixel bias map values directly into the telemetry buffer using *biasHuffman*.packData().
11. If the buffer becomes full, the maximum allowed rows for one telemetry packet has been packed, or the last row of the bias image has been processed, trickleTeBias() posts the telemetry packet buffer to telemetry, using *form*.post(). If there are more rows to process, it obtains a new buffer for the form, and repeats the packing operation. Once the entire bias map for the FEP has been processed, trickleTeBias() returns to its caller. The caller may then re-invoke trickleTeBias() for another FEP's map. This continues until all of the maps have been posted, or until the operation is aborted.

38.4.4 Use 4: Abort transmission of the pixel bias maps

In order to stop the bias trickle algorithm before it completes, the client calls *biasThief*.abort(). This sets the thief's abort flag, and sends an event notifying it that the operation has been aborted, using *biasThief*.notify(). If the operation has already completed, then the notification is consumed in the main loop and ignored. If the operation is waiting for a telemetry packet buffer, the notification is consumed by the call to requestEvent(), and the transmission of the current map, and maps from subsequent FEPs is aborted.

38.5 Class BiasThief

Documentation:

This class is responsible for copying the pixel bias map values from the Front End Processors, and trickling these maps to telemetry.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **`Task`**

Implementation Uses:

`Tf_Data_Te_Bias_Map`
`Tf_Data_Cc_Bias_Map`
`TaskMonitor` *`taskMonitor`*
`HuffmanMap` *`huffmanMap`*

Public Interface:

Operations:

```
BiasThief()
abort()
biasReady()
goTaskEntry()
selectMode()
setFepInfo()
```

Protected Interface:

Operations:

```
checkMonitor()
getBuffer()
setupTeForm()
trickleCcBias()
trickleTeBias()
```

Private Interface:

Constants: PIXELS_PER_ROW = 1024 locations
BUFFER_TIMEOUT = 1 second
MAXROWS_PER_PACKET = 10 rows

Has-A Relationships:

unsigned *timestamp*: This is a copy of the start time of the bias run.

unsigned *blockid*: This is a copy of the parameter block id used to start the run.

unsigned *modetype*: This is a copy of the type of bias map being telemetered. 0 indicates Timed-exposure, and 1 indicates Continuous Clocking.

Boolean *abortFlag*: This indicates whether the bias has been aborted or not. This flag is cleared by calls to `biasReady`() and is asserted by calls to `abort`().

static HuffmanTable *biasHuffman*: This is the Huffman Compression table object used by the Bias Thief to compress bias map data into telemetry packet buffers.

const unsigned *buffer_timeout*: This is the maximum number of timer ticks to wait for a telemetry buffer before checking for task monitor queries, or abort requests (Acis::TICKS_PER_SECOND).

const unsigned *maxrows*: This is the maximum number of rows that should be packed into a single telemetry buffer (10 TBD).

const unsigned *pixels_per_row*: This is the number of pixels locations in 1 row in the bias map (1024).

struct FepInfo *fepInfo*[6]: This is an array of information structures used to configure the bias telemetry operation for each FEP. The structure is as follows:

const unsigned short* *base*: Points to FEP's pixel bias map. 0 if FEP unused.

CcdId *ccd*: Speicifes which CCD the FEP is processing

unsigned *rowpixels*: Number of pixels in 1 bias map row

unsigned *rowcnt*: Number of rows in bias map

unsigned *scale*: Number of CCD rows summed on-chip into image row

unsigned *biasoffset*[4]: Pixel bias map offsets for each quadrant

unsigned *compress*: Compression table selection for the FEP

Concurrency: Active

Persistence: Persistent

38.5.1 BiasThief()

Public member of: **`BiasThief`**

Arguments:

`unsigned` *`taskid`*

Documentation:

This is the constructor for the Bias Thief task. *`taskid`* is the RTX identifier for the task. This function passes *`taskid`* to its parent's constructor, `Task`(), and initializes the constants used by the class.

Concurrency: Sequential

38.5.2 abort()

Public member of: **`BiasThief`**

Return Class: **`void`**

Documentation:

This function causes the bias thief to abort its current bias telemetry processing. This function sets *`abortFlag`* to *`BoolTrue`* and calls `notify`() to signal the task that an abort has been requested.

Concurrency: Synchronous

38.5.3 biasReady()

Public member of: **BiasThief**

Return Class: **void**

Documentation:

This function indicates that the bias maps for all of the enabled Front End Processors are ready to be telemetered. This function sets *abortFlag* to *BoolFalse*, and then uses notify() to inform the task that a bias map is ready to be sent.

Preconditions:

The client must first call selectMode(), and must then call setFepInfo() for each FEP in use. This is only required after a reset, and if the current mode or FEP parameters change. (Although it is not required by the class, it is strongly recommended to call these functions for each run).

Concurrency: Synchronous

38.5.4 checkMonitor()

Protected member of: **BiasThief**

Return Class: **Boolean**

Documentation:

This function responds to queries from the task monitor, and detects whether or not the current operation has been aborted. If the operation has been aborted, the function returns *BoolFalse*. If not, it returns *BoolTrue*.

Semantics:

This function uses requestEvent() to poll for the task monitor query or an abort signal. If the task monitor has issued a query, this function uses *taskMonitor*.respond() to respond to query. If an abort signal has been issued, the function returns *BoolFalse*, otherwise, it returns *BoolTrue*.

Concurrency: Synchronous

38.5.5 getBuffer()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

TlmForm& *form*

Documentation:

This function waits for and allocates a buffer for the telemetry format, *form*, while responding to queries from the *taskMonitor*. If successful, the function returns *BoolTrue*. If the operation is aborted, the function returns *BoolFalse*.

Semantics:

This function consists of a loop which acquires a telemetry buffer for *form*. by passing *buffer_timeout* to *form*.waitForBuffer(). If a buffer is allocated, the function returns. If the wait times out, the function calls checkMonitor() to respond to any task monitor queries, and to detect an abort of the bias telemetry operation. If aborted, the function returns immediately. If not aborted, the loop iterates.

Concurrency: Synchronous

38.5.6 goTaskEntry()

Public member of: **`BiasThief`**

Return Class: **`void`**

Documentation:

This function contains the main loop of the bias thief task.

Semantics.

This function consists of an infinite loop. At the top of the loop, the function waits for and consumes start, abort and task monitor query signals. Abort signals are discarded. If a task monitor query signal is received, `goTaskEntry`() responds using *`taskMonitor`*`.respond`(). If a start signal is received, and *`abortFlag`* is *`BoolFalse`*, `goTaskEntry`() enters a loop which sends the bias maps for each FEP in the system. (NOTE: If *`abortFlag`* is *`BoolTrue`*, the abort request occurred after the start request, and no bias telemetry operation should be attempted). Within the loop, if the base address for a given FEP is zero, then the corresponding FEP's bias is not sent, and the loop skips to the next FEP (see `selectMode`() and `setFepInfo`()). If *`modetype`* is 0, then a Timed Exposure bias map is being sent, and the function calls `trickleTeBias`() to send the map. Otherwise, it is a Continuous Clocking bias, and `goTaskEntry`() calls `trickleCcBias`() instead. If either function returns *`BoolFalse`*, then an abort request has been received, and the bias telemetry operation from subsequent FEPs is aborted. Otherwise, the process repeats for each used FEP.

Concurrency: Synchronous

38.5.7 selectMode()

Public member of: **BiasThief**

Return Class: **void**

Arguments:

unsigned *mode*
unsigned *starttime*
unsigned *parameterId*

Documentation:

This function selects whether or not to dump Timed-Exposure bias values, or Continuous Clocking bias values, and configures the start time of the bias run, and the parameter block id used to produce the map. If *mode* is 0, Timed Exposure is used. If *mode* is 1, Continuous Clocking is used. *starttime* indicates the latched ACIS science timestamp at the start of the bias run, *parameterId* is the id from within the parameter block used to configure the run.

Preconditions:

A bias telemetry operation must not already be in progress.

Semantics:

This function saves the passed parameters in its instance variables, and then flags all FEPs as unused by zeroing the bias base address for each FEP entry in the *fepInfo*[] array.

Postconditions:

No FEPs are configured to be used. For each FEP to be used, setFepInfo() must be called to set the bias map telemetry parameters for the corresponding FEP.

Concurrency: Synchronous

38.5.8 setFepInfo()

Public member of: **BiasThief**

Return Class: **void**

Arguments:

FepId *fepid*
const unsigned short* *base*
CcdId *ccd*
unsigned *rowpixels*
unsigned *rowcnt*
const unsigned *biasoffset[4]*
unsigned *compress*
unsigned *scale*

Documentation:

This function sets up the bias thief to steal pixel bias values from a particular FEP, indicated by *fepid*. *base* is the base address, within the FEP, of the bias map. *ccd* indicates which CCD produce the map. *rowpixels* is the number of map values in each row, and *rowcnt* is the number of rows from the map to telemeter. The *biasoffsets* array contains the pixel initial overclocks for each video chain. *table* specifies which compression table to use (ignored if continuous clocking is being performed). *scale* is the number of CCD rows in each image row (i.e. in 2x2 summing, there are two CCD rows summed into 1 image row).

Preconditions:

A bias telemetry operation must not already be in progress, and `selectMode`() must have been called.

Semantics:

This function saves the passed parameters in the *fepInfo*[] entry indexed by *fepid*.

Postconditions:

Once started, the bias map from *fepid* will be telemetered.

Concurrency: Synchronous

38.5.9 setupTeForm()

Protected member of: **BiasThief**

Return Class: **void**

Arguments:

Tf_Data_Te_Bias_Map& *form*
unsigned *packetNum*
unsigned *pixelrow*
FepId *fepid*

Documentation:

This function sets up the Timed Exposure telemetry form, *form*. *packetNum* is the packet number in the series for this FEP, and *pixelrow* is the row number, in image coordinates, of the first row in the packet. Since data is packed last row to first, subsequent rows in the packet have decrementing positions. *fepid* is the id of the FEP whose bias map is being sent.

Preconditions:

The form must have allocated a telemetry packet buffer.

Semantics:

This function uses the *form* to store the passed information into the telemetry packet buffer, and zeros the bias data length using *form*.setEmpty().

Concurrency: Synchronous

38.5.10 trickleCcBias()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function trickles the Continuous Clocking bias map from the FEP indicated by *fepid*. If successful, the function returns *BoolTrue*. If it is aborted, it returns *BoolFalse*.

Semantics:

This function first calls yield() to allow other tasks of the same priority to run. It then calls checkMonitor() to respond to any task monitor queries, and to check for any abort requests. If aborted, trickleCcBias() returns immediately. If not aborted, it proceeds to send the bias. It first passes a NULL table pointer to *biasHuffman*.loadTable() to configure the data compression algorithm to bit-pack the data, without compressing it, and *biasHuffman*.reset() to initialize the state of the packing algorithm. It then creates a bias telemetry form, *form*, and calls getBuffer() to obtain a telemetry packet buffer. If getBuffer() indicates an abort, the function returns immediately. Once a buffer has been obtained, the function zeros the bias data length of the buffer, and uses the form to set the start time, parameter block id, CCD Id, FEP Id, and initial overclocks into the telemetry buffer. It then uses *form*.get_Data_Address() and form.get_Data_Avail() to get the bias data buffer address and length within the telemetry packet buffer. It then passes these to *biasHuffman*.packData() to pack the one continuous clocking bias map row into the telemetry packet buffer. It calls *form*.set_Data_Written() to set the bias data word count in the buffer, and then uses *form*.post() to post the packet buffer to telemetry.

Concurrency: Synchronous

38.5.11 trickleTeBias()

Protected member of: **BiasThief**

Return Class: **Boolean**

Arguments:

FepId *fepid*

Documentation:

This function trickles the Timed Exposure bias map from the FEP indicated by *fepid*. If successful, the function returns *BoolTrue*. If it is aborted, then it returns *BoolFalse*.

Semantics:

This function uses *huffmanMap*.getTable() to obtain the table pointer for the FEP's compression selection, and passes this pointer to *biasHuffman*.loadTable(). It then initializes some local variables and enters its row processing loop. Rows are processed in reverse order, from the end of the bias map to the beginning.

On each iteration, the function calls yield() to allow other tasks of the same priority to run, and then checkMonitor() to respond to task monitor queries and detect abort requests. If aborted, the function returns immediately. The loop checks to see of the telemetry form, *form*, has a buffer using *form*.hasBuffer(), and if not, calls getBuffer() to allocate a buffer, setupTeForm() to initialize its contents, and *biasHuffman*.reset() to reset the compression state.

It then calls *biasHuffman*.packData() to pack one row of bias data to the end of the telemetry buffer. If the entire row fit, it updates its row information. If the row does not fit into the buffer, or if the last row in the map has been packed, or if the maximum number of rows per packet have been put into the telemetry buffer, the function uses *form* to store the total number words written, and the total number of rows written into the telemetry buffer. It then uses *form*.post() to post the buffer to telemetry.

NOTE: If the *form* has a telemetry buffer when an abort is detected, the destructor for the form will release the buffer back into its pool. This prevents the abort causing buffers to be "lost."

Concurrency: Synchronous

39.0 FEP IO Library (36-53223 B)

39.1 Purpose

The FEP IO Library is a set of functions, executing in the FEP, which provide an interface to the front end hardware. Clients may use these functions to direct the FEP without having to know low level hardware details.

39.2 Uses

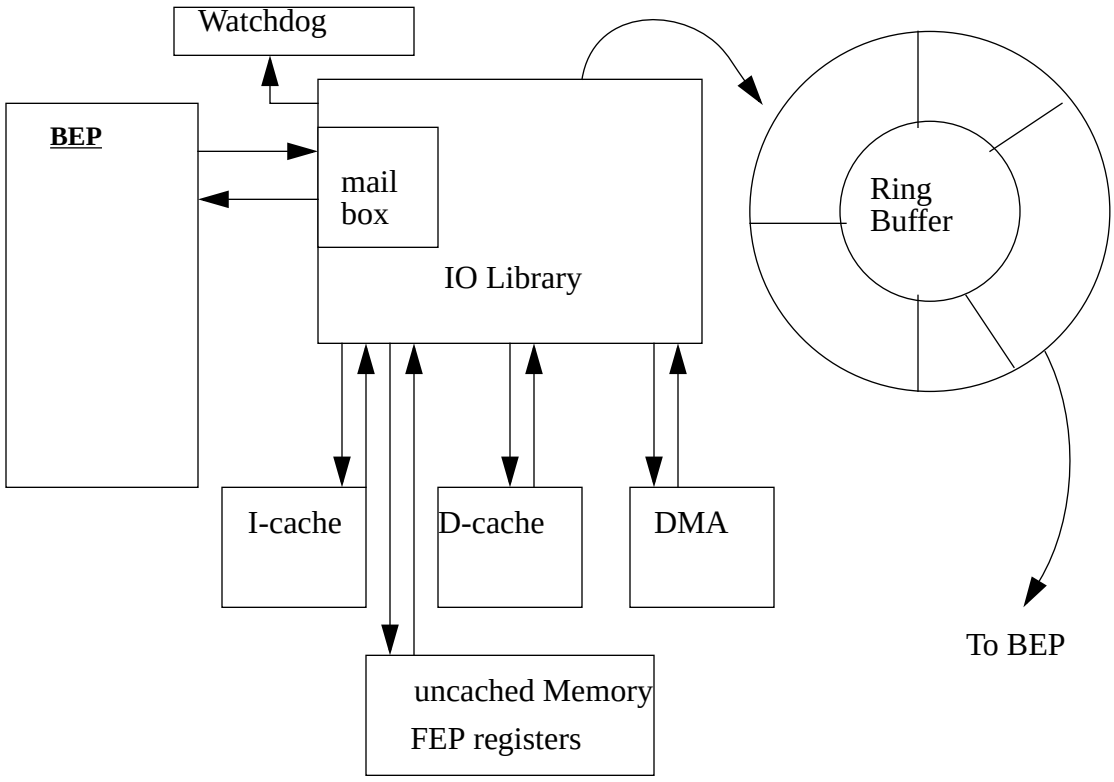
The FEP IO Library provides the following features:

Use 1:: Transfer data from the FEP to the BEP
Use 2:: Handle communications between the FEP and BEP
Use 3:: Manage the Mongoose DMA
Use 4:: Provide read/write access to internal Mongoose memory and status words
Use 5:: Reset the Watchdog Timer

39.3 Organization

Figure 186 illustrates the interfaces to the FEP provided by the IO Library.

FIGURE 186. FEP IO Library Interface Diagram



The FEP IO Library provides access functions to various areas of the FEP and BEP hardware.

- BEP/FEP - There are several aspects to the interface between the BEP and the FEP. The ring buffer is in shared memory. The FEP deposits data (usually image data, histograms or photon events) into the buffer in blocks less than or equal to 32-long words, which are in turn read out by the BEP (NOTE: If a given piece of data is less than 32-words in length, it is padded out to the full 32-word block size in the ring buffer).

  Another aspect of the interface is the communication protocol between the BEP and FEP. The method used is by a rudimentary mailbox. The BEP writes commands to the mailbox, and looks for replies in the same box.

  See the **ACIS Software Requirements Specification (#36-41001-02)** and the **ACIS Science Instrument Software Preliminary Design Specification (#36-02402)** for further details on the BEP/FEP hardware and software interface.

- ***FEP Internal*** - The other main function of the IO Library is to provide access to or control of the Mongoose internal memory. It is necessary to access the D-cache and I-cache, allow a method for executing a function in memory, and also to control the DMA for data transfers. There are also a set of basic access routines to some of the FEP hardware registers. A detailed description of the hardware registers and FEP memory can be found in the **DPA Hardware Specification & System Description #36-02104**.

- ***Watchdog*** - IO library also provides a function for maintaining the FEP watchdog timer. The timer must be reset periodically to avoid a hardware reset.

  See the **ACIS Software Requirements Specification (#36-41001-02)** and the **ACIS Science Instrument Software Preliminary Design Specification (#36-02402)** for further details on the watchdog timer.

39.4 Scenarios

The following paragraphs describe the basic function performed by the FEP, and which library functions should be used in a given scenario. In general, the functions prefaced by FIO (upper case) should be adequate for any task required by the science processing. All function prefaced with a lower case (fio) are used at the directive of the BEP.

39.4.1 Use 1: Transfer data from the FEP to the BEP

Transferring data across the interface uses a portion of the shared memory which has been configured as a ring buffer, consisting of a series of 32-long word blocks. The function **`FIOappendBlock()`** writes the specified data block to the ring buffer. This function does not return until the operation is completed, or until a command arrives at the BEP to FEP Command Mailbox.It is possible for the buffer to be full, so a wait may be required until the BEP has emptied more of the buffer. While waiting for space in the buffer, this function calls **FIOtouchWatchdog()**. If the BEP is too slow in clearing the buffer, science processing may be affected. The function **`fioRbStatus()`** returns the status of the ring buffer and the available number of blocks. The library users do not need to call this routine before calling **`FIOappendBlock()`**.

39.4.2 Use 2: Handle communications between the FEP and BEP

The FEP has little independent action; therefore, there is a command protocol for directing the FEP. A portion of the shared memory is set aside for a mailbox. The function **`FIOgetNextCmd()`**polls the command mailbox to look for pending commands and performs the required actions. The functions **`fioPollMBox()`**, **`fioGetCmd()`**, and **`FIOwriteCmdReply()`**are used by **`FIOgetNextCmd()`**to handle the command interface. If the command is directed to the science processing, it returns that information to the calling function.

39.4.3 Use 3: Manage the FEP DMA

The Mongoose DMA may be used to transfer data between the D-cache and general purpose memory, or between two general memory locations. The function **`FIOdmaTransfer()`**initiates the transfer. The function **`FIOdmaDone()`**indicates when the transfer has completed.

39.4.4 Use 4: Provide read/write/execute access to internal FEP memory

The I-cache requires special access through read/write registers. The functions **`fioReadIcache()`**and **`fioWriteIcache()`** should be used to for any access to the I-cache. For completeness, the functions **`fioReadMem()`**and **`fioWriteMem()`**are available for accessing the D-cache. There are also several functions available to read and/or write some of the FEP hardware registers and access the addresses of the various por-

tions of hybrid memory, this includes the capability for directly executing a function located in the memory.

39.4.5 Use 5: Reset WatchdogTimer

The watchdog timer must be reset frequently. If the timer reaches zero, it assumes the data processing is hung and resets the FEP. The function **`FIOtouchWatchdog()`** should be called within a **TBD** interval to reset the timer.

See the ACIS Software Preliminary Design Specification (FEP IO Library), (#36-02402) for further details on this interface.

39.5 BEP - FEP Communication Protocol

Communication between the BEP and the FEP is accomplished via two mailboxes, one for BEP initiated commands and their corresponding responses, the other for FEP initiated requests and their corresponding replies. The mailboxes are located in a section of shared memory between the BEP and the FEP. It is necessary to have some handshaking to ensure proper delivery and receipt of messages.

39.5.1 Mailbox Structure

The first field contains the mailbox state, the second contains the message. The actual message is divided into further sub-fields: the length of the following message, the type field, which indicates the kind of message, and any additional arguments required. Replies are written into the same structure as the initial command or request. (NOTE: See “fio.h” for a formal definition of this structure, and the definition of SIZE).

Command Mailbox

```
typedef struct {
  MBOXstate state;
  typedef struct {
      unsigned len;
      int type;
      unsigned args[SIZE];
  } COMMAND;
} CMD_MBOX;
*mailbox size is still 128 32-bit
words
```

39.5.2 BEP to FEP

Communication originating in the BEP is usually a command directive to the FEP. There are two general types of commands, diagnostic and science. Diagnostic commands require information regarding the state of the FEP and are handled by the IO library. All science commands are forwarded to the science process.

The transfer sequence is as follows: the BEP writes a command to the mailbox and changes the state to NEW_MESSAGE. The FEP has been polling the mailbox (**`FIOgetNextCmd()`**) and takes the appropriate action when a command is received. In all cases, a response must be written to the command by the recipient (**`FIOwriteCmdReply()`**). This function will write the response and set the mailbox state to REPLY_RDY. The BEP reads the response and sets the mailbox state to NO_MESSAGE in preparation for the next exchange.

39.5.2.1 Diagnostic Command Formats

The following is a list of the commands that the FEP responds to. All other commands are forwarded to the science process.

39.5.2.1.1 Read I-Cache

This command accepts a source I-cache address and count of words to be read, in the format specified by Table 30 . If the parameters are within valid ranges, the data are read from I-Cache and stored back in the mailbox in the format specified by Table 31 . If the parameters are invalid, the action taken by the FEP is **TBD**.

TABLE 30. Read I-Cache command format

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	2	Number of command arguments + type
type	Command type	enum	CMD_READ_ICACHE	Execute Read I-Cache command
args[0]	Address	32 bit unsigned	0x80080000 - 0x800ffffe	Source address in I-cache to read data from.
args[1]	Length	32 bit unsigned	0 -127	Number of words to be read. May not cross I-cache boundary

TABLE 31. Read I-Cache command response

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	0 - 127	Number of words in arguments field + type
type	Command type	enum	CMD_READ_ICACHE	Indicates which command has been executed
args[0 .. N]	Data	32 bit unsigned		Data values read from I-cache.

39.5.2.1.2 Write I-cache

This command accepts a destination address, count of words to be written, and data, in the format specified by Table 32 . If the parameters are within valid ranges, the data are writ-

ten into I-Cache. The command response is detailed in Table 33 . If the parameters are invalid, the action taken by the FEP is **TBD**.

TABLE 32. Write I-Cache command format

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	0 - 127	Number of command arguments + type
type	Command type	enum	CMD_WRITE_ICACHE	Execute Write I-Cache command
args[0]	Address	32 bit unsigned	0x80080000 - 0x800ffffe	Destination address in I-cache to write data too.
args[1.. N]	Data	32 bit unsigned		Data values to write to I-cache.

TABLE 33. Write I-Cache command response

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	1	Number of command arguments + type
type	Command type	enum	CMD_WRITE_ICACHE	Indicates which command has been executed

39.5.2.1.3 Read Memory

This command accepts a source address and count of words to be read, in the format specified by Table 34 . If the parameters are within valid ranges, the data are read from D-cache or general memory and stored back in the mailbox in the format specified by Table 35 . If the parameters are invalid, the action taken by the FEP is **TBD**.

TABLE 34. Read memory command format

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	2	Number of command arguments + type
type	Command type	enum	CMD_READ_MEM	Execute read memory command
args[0]	Address	32 bit unsigned	0x80000000 - 0x8007fffc 0xa0000000 - 0xfffffffc	Source address in memory to read data from.
args[1]	Length	32 bit unsigned	0 - 127	Number of words to be read. May not memory boundaries

TABLE 35. Read memory command response

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	0 - 127	Number of words in arguments field + type

TABLE 35. Read memory command response

Argument	Field	Units	Values	Description
type	Command type	enum	CMD_READ_MEM	Indicates which command has been executed
args[0 .. N]	Data	32 bit unsigned		Data values read from memory

39.5.2.1.4 Write Memory

This command accepts a destination address, count of words to be written, and data, in the format specified by Table 36 . If the parameters are within valid ranges, the data are written into memory. The command response is detailed in Table 37 . If the parameters are not valid, the action taken by the FEP is **TBD**.

TABLE 36. Write memory command format

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	0 - 127	Number of command arguments + type
type	Command type	enum	CMD_WRITE_MEM	Execute write memory command
args[0]	Address	32 bit unsigned	0x80000000 - 0x8007fffc 0xa0000000 - 0xfffffffc	Destination address in write data to.
args[1.. N]	Data	32 bit unsigned		Data values to write to memory.

TABLE 37. Write memory command response

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	1	Number of command arguments + type
type	Command type	enum	CMD_WRITE_MEM	Indicates which command has been executed.

39.5.2.1.5 Execute Memory

This command accepts the address of a function and a list of the functions arguments, in the format specified by Table 38 . The function is called with the indicated arguments. the return value of the function is stored back in the mailbox in the format specified by Table 39 .

TABLE 38. Execute memory command format

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	0 - 20	Number of command arguments + type
type	Command type	enum	CMD_EXECUTE_MEM	Execute “execute memory” command

TABLE 38. Execute memory command format

Argument	Field	Units	Values	Description
args[0]	Address	32 bit unsigned		Pointer to function to be called.
args[1..20]	Arguments	32 bit unsigned		Arguments for function call.

TABLE 39. Execute memory command response

Argument	Field	Units	Values	Description
len	Argument count	32 bit unsigned	2	Number of words in arguments field + type
type	Command type	enum	CMD_EXECUTE_MEM	Indicates which command has been executed.
args[0]	Data	32 bit unsigned		Return value from function call.

39.6 Specification

The following are the function definitions for the FEP IO Library. There are several access functions that will just return an address pointer, a hardware register value, or a constant. Most of these will be implemented as macros or in-line functions, defined in the file fep.h. See the **DPA Hardware Specification & System Description #36-02104, section TBD,** for more detailed information.

The functions defined in Section 39.6.1 through Section 39.6.19 are interface functions to read and write the FEP hardware registers. These are defined as in-line functions in the include file fep.h.

39.6.1 FIOgetBiasMapPtr()

Scope: Science

Return type **`unsigned *`**

Return a pointer to the start of the bias map. The address points to a pre-defined array of 1024 x 1024 16-bit pixels, and begins on a 4-byte boundary in non-cache memory.

39.6.2 FIOgetBiasConfig()

Scope: Science

Arguments:

unsigned *`*biasrec`*

unsigned *`cnt`*

Return type **`void`**

Retrieve the Bias Configuration information stored in I-Cache. *`cnt`* words are read from the I-cache address specified by the constant BIAS_CONFIG_SAVE and stored in the buffer indicated by *`biasrec`*.

39.6.3 FIOsetBiasConfig()

Scope: Science

Arguments

unsigned *`*biasrec`*

unsigned *`cnt`*

Return type **`void`**

Write *`cnt`* words of bias configuration data specified by *`biasrec`* into I-cache. The location in I-cache is specified by the constant BIAS_CONFIG_SAVE.

39.6.4 FIOgetBiasParityPlanePtr()

Scope: Science

Return type **`unsigned *`**

Return a pointer to the location of the bias parity plane previously specified by a call to **`fioSetSegAllocReg()`**.

39.6.5 FIOgetCcdRowStart()

Scope: Science

Return type **`unsigned`**

Return the row index of the CCD where processing should begin, not necessarily the first row. The first row has index 0.

39.6.6 FIOsetCcdRowStart()

Scope: Science

Arguments

unsigned *`row_index`*

Return type **`void`**

Set the index row of the CCD where processing should begin. The first row has index 0.

39.6.7 FIOgetImageMapPtr()

Scope: Science

Return type **`unsigned *`**

Return a pointer to the start of the Image map. The address points to a pre-defined array of 1024 x 1024 16-bit pixels, and begins on a 4-byte boundary in non-cache memory.

39.6.8 FIOgetImageMapRowIndex()

Scope: Science

Return type **unsigned**

Return an index to the last row written in the image map. The first map has index 0.

39.6.9 FIOgetImageMapRowLength()

Scope: Science

Return type **unsigned**

Return the count of CCD rows.

39.6.10 FIOsetImageMapRowLength()

Scope: Science

Arguments

unsigned *count*

Return type **void**

Set the number of CCD rows.

39.6.11 FIOgetImageMapRowStart()

Scope: Science

Return type **unsigned**

Return the Image Map row index at which the hardware should begin loading the image into non-cache memory. The first row has index 0.

39.6.12 FIOsetImageMapRowStart()

Scope: Science

Arguments

unsigned *row_index*

Return type **`void`**

Set the index row of the Image Map where we should begin loading the image. The first row has index 0.

39.6.13 FIOgetStartCntReg()

Scope: Science

Arguments

unsigned *`regnum`*

Return type **`unsigned`**

Return the contents of the indicated register (0 - 3).

39.6.14 FIOsetStartCntReg()

Scope: Science

Arguments

unsigned *`regnum`*

short *`value`*

Return type **`void`**

Set the indicated offset register (0 - 3) to the value specified.

39.6.15 FIOgetOverclockBufPtr()

Scope: Science

Return type **`unsigned *`**

Return a pointer to the start of the overclock buffer previously specified by a call to **`fioSetSegAllocReg()`**.

39.6.16 FIOgetThresholdRegister()

Scope: Science

Arguments

unsigned *regnum*

Return type **short**

Return the contents of the indicated threshold register (0 - 3).

39.6.17 FIOsetThresholdRegister()

Scope: Science

Arguments

unsigned *regnum*

short *thresholdval*

Return type **void**

Set the indicated threshold register (0 - 3) to the value specified. In order to support updating the threshold register values synchronously with the advent of the next exposure, this function stores the passed thresholdval into a holding variable. When the Beginning of Frame interrupt occurs, the interrupt handling routine will copy the contents of the holding variable into the actual threshold register.

NOTE: This scheme requires that the interrupt handler is capable of copying the holding variables into the threshold registers within one pixel clock of the Beginning of Frame interrupt (~10μseconds).

39.6.18 FIOgetThresholdXings()

Scope: Science

Return type **unsigned**

Return the value of the T-Plane crossings counter.

39.6.19 FIOgetTPlanePtr()

Scope: Science

Return type **`unsigned *`**

Return a pointer to the start of the T-Plane, previously specified by a call to **`fioSetSeg-AllocReg()`**.

39.6.20 FIOappendBlock()

#include fio.h

Scope: Science

Return type: **bool**

Arguments

unsigned ***ptr**

unsigned *wordcnt*

Description:

Copy *wordcnt* words (32 bit values) pointed to by *ptr* to the block at the end of the ring buffer. *wordcnt* must be less than or equal to 32. Although *wordcnt* may be less than 32 words, the written block will always consume 32-words of ring-buffer space. This function does not return until the action is completed, or until a command is received at by the Command Mailbox. There may be delays if the ring buffer is full. Data values must be aligned on long word boundaries. While waiting for room the function maintains the watchdog timer using **FIOtouchWatchdog()**, and polls the status of the Command Mailbox using **fioPollMBox()**. If the block is successfully appended to the ring buffer, the function returns TRUE. If a command arrives while waiting for room in the ring buffer, the function aborts, and returns FALSE. Using **FIOgetNextCmd()**,the caller must then read and process the received command.

NOTE: Interrupts to the BEP on writes to an empty ring-buffer have been replaced by having the BEP poll the ring-buffers for data.

39.6.21 FIOdmaDone()

#include fio.h

Scope: Science

Return type **bool**

Arguments: none

Description:

This function returns TRUE if the most recent DMA transfer has completed. It returns FALSE if the transfer is not complete.

39.6.22 FIOdmaTransfer()

#include fio.h

Scope: Science

Return type **`void`**

Arguments:

unsigned * *src*

unsigned * *dest*

unsigned *wordcnt*

Conditions:

This function requires a mongoose processor to be tested, it can not be tested on a development workstation.

Description:

Transfer *`wordcnt`* words indicated by *`src`* across the DMA interface to location indicated by *`dest`*. This operation can only be used between D-cache and memory, and memory to memory. If *`src`* and *`dest`* both point to D-cache, a panic will be generated.

39.6.23 FIOgetExpInfo()

#include fio.h

Scope: Science

Return type: **`void`**

Arguments:

unsigned *`expnum`

unsigned *`timestamp`

Conditions:

This function requires a mongoose processor to be tested, it can not be tested on a development workstation.

Description:

Return information about the current exposure, i.e., the exposure number and the timestamp of the exposure. Interrupts will be masked while this function executes. The interrupts will be disabled for less than ~5μseconds.

39.6.24 FIOgetNextCmd()

#include fio.h

Scope: Science

Return type: **`bool`**

Arguments

COMMAND *`cmd`

Description:

Returns TRUE if a new science command has arrived, in which case, structure `cmd` contains the science command. Returns FALSE, and leaves `cmd` unchanged, if there are no new science commands. If the command is diagnostic in nature, this functions does the necessary processing before returning. Refer to the file “fio.h” for a list of possible diagnostic command types. All diagnostic command types are negative in value. Non-diagnostic command types, which are not defined in “fio.h”, must have a value greater than 0.

39.6.25 FIOinit()

#include fio.h

Scope: Science

Return type: **`void`**

Arguments none

Description

This function initializes the various hardware registers in the FEP. The ring buffer is initialized by the BEP. Refer to the DPA Hardware Specification & System Description for a detailed accounting of the registers and initialization information.

39.6.26 FIOtouchWatchdog()

#include fio.h

Scope: Control

Return type: **`void`**

Arguments: none

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function resets the watchdog timer to prevent a hardware reset.

39.6.27 FIOwriteCmdReply()

#include fio.h

Scope: Control

Return type: **void**

Arguments:

COMMAND* *cmdreply*

Description

This function writes the command into the command reply buffer. It is required after a call to **FIOgetNextCmd()** returns the value TRUE, and is also called with **FIOgetNextCmd()** to respond to diagnostic commands.

39.6.28 fioClearBitCtrlReg()

#include fio.h

Scope: Diagnostic

Return type **void**

Arguments:

unsigned *mask*

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function clears the bits in the FEP Control Register indicated by the *mask*. It does not affect any of the other bits in the control register. The constants for setting particular bits are defined in the enum CtlRegMask in fio.h.

39.6.29 fioClearBitImCtrlReg()

#include fio.h

Scope: Diagnostic

Return type **`void`**

Arguments:

unsigned *`mask`*

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function clears the bits in the FEP Image Control Register indicated by the *`mask`*. It does not affect any of the other bits in the register. The constants for setting particular bits are defined in the enum ImgCtlRegMask in fio.h.

39.6.30 fioGetCmd()

#include fio.h

Scope: Control

Return type: **`void`**

Arguments:

COMMAND * *`cptr`*

Conditions:

This function does not check if the command is current. Use **`fioPollMBox()`** to check the status of the command.

Documentation

Get a command from the BEP. The command is returned in the area indicated by *`cptr`*.

39.6.31 fioGetStatusReg()

#include fio.h

Scope: Diagnostic

Return type **`unsigned`**

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function returns the value of the status register.

39.6.32 fioGetImStatusReg()

#include fio.h

Scope: Diagnostic

Return type **`unsigned`**

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function returns the value of the Image Status register.

39.6.33 fioPollMBox()

#include fio.h

Scope: Control

Return type: **`bool`**

Arguments:

`MBOXstate*mbox`

MBOXstate *`state`*

Documentation

Returns TRUE if the MBOXstate of the mailbox matches *`state`* in the argument list. Returns FALSE otherwise. Mailbox state variables are defined in “fio.h”.

39.6.34 fioRbStatus()

#include fio.h

Scope: Science

Return type:

RINGBUFstatus

Arguments:

unsigned *`available`

Description

This function returns the status of the ring buffer. This function should be called before any other functions concerning the ring buffer are used. `Available` contains the number of 32-word blocks available in the buffer.

Return Values

RB_FULL - Ring buffer is full, do not append more data

RB_EMPTY - Ring buffer is empty

RB_NOT_FULL - Ring buffer is not full and not empty, check `available` to see if there is enough room to append more data.

RB_ERROR - There is an internal error in the ring buffer

39.6.35 fioReadIcache()

#include fio.h

Scope: Diagnostic

Return type: **void**

Arguments:

unsigned * *src*

unsigned * *dest*

unsigned *wordcnt*

Conditions:

This function requires a mongoose processor to be tested, it can not be tested on a development workstation.

Description

This function reads *wordcnt* 32-bit words from the I-cache, starting at *src*, into the buffer indicated by *dest*.

39.6.36 fioReadMem()

#include fio.h

Scope: Diagnostic

Return type **void**

Arguments:

unsigned * *src*

unsigned * *dest*

unsigned *wordcnt*

Description

This function reads *wordcnt* 32-bit words starting at *src* in D-cache or general memory into the buffer indicated by *dest*.

39.6.37 fioSetBitCtrlReg()

#include fio.h

Scope: Diagnostic

Return type **void**

Arguments:

unsigned *mask*

Description

This function sets the bits in the Control register indicated by *mask*. It does not affect any of the other bits in the register. The constants for setting particular bits are defined in the enum CtrlRegMask in the file fio.h.

39.6.38 fioSetBitImCtrlReg()

#include fio.h

Scope: Diagnostic

Return type **void**

Arguments:

unsigned *mask*

Description

This function sets the bits in the Image Control register indicated by *mask*. It does not affect any of the other bits in the register. The constants for setting particular bits are defined in the enum ImgCtrlRegMask in the file fio.h.

39.6.39 fioSetSegAllocReg()

#include fio.h

Scope: Control

Return type **void**

Arguments:

unsigned *pplane*
unsigned *tplane*
unsigned *oclk*

Description

This function sets the bits in the Bulk Memory Segment Allocation register. It is used to designate where in bulk memory the parity plane, threshold plane, and overclock buffer are located. It does not affect any of the other bits in the register. *pplane*, *tplane*, and *oclk* specify which segment to use for the parity plane, threshold plane, and overclock memory, respectively.Each must be unique and may range from 0 to 15. Each increment selects a 128Kbyte block within bulk memory, where 0 corresponds to the first block within bulk memory. The constants used for setting particular bits are defined by the enum SEGALLOC in fio.h.

39.6.40 fioWriteIcache()

#include fio.h

Scope: Diagnostic

Return type: **void**

Arguments:

unsigned * *src*

unsigned * *dest*

unsigned *wordcnt*

Conditions:

This function requires a mongoose processor to be tested, it can not be tested on a development workstation.

Description

This function writes *wordcnt* 32-bit words starting at *src* into the I-cache starting at *dest.*

39.6.41 fioWriteMem()

#include fio.h

Scope: Diagnostic

Return type: **void**

Arguments:

unsigned * *src*

*unsigned * dest*

unsigned *wordcnt*

Description

This function writes *wordcnt* 32-bit words starting at *src*, into the D-cache or general memory starting at *dest*.

39.6.42 fioWritePulseReg()

#include fio.h

Scope: Diagnostic

Return type **void**

Arguments:

unsigned *mask*

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function sets the bits in the Pulse register indicated by *mask*. It does not affect

any of the other bits in the register. The constants for setting particular bits are by the enum PulseRegMask in fio.h.

39.6.43 fioWriteImPulseReg()

#include fio.h

Scope: Diagnostic

Return type **`void`**

Arguments:

unsigned *`mask`*

Conditions:

This function requires actual ACIS hardware to be tested, it can not be tested on a development workstation.

Description

This function sets the bits in the Image Pulse register indicated by *`mask`*. It does not affect any of the other bits in the register. The constants for setting particular bits are by the enum ImgPulseRegMask in fio.h.

40.0 Boot FEP (36-53230 A)

40.1 Purpose

The FEP is booted at the direction of a BEP function which commands and controls the process. The Boot FEP executable runs on the FEP. The FEP Manager function `loadRunProgram` might install `fepCtl` and `fepTimedBias` et. al. to initiate a science run. Refer to the FEP Manager *Section 25.5.2 Use 2: Reset, load, and run a program on a Front End Processor*.

Boot code is copied by the BEP into shared FEP bulk memory[1]. The boot code executing in the FEP bulk memory, will provide the BEP with the capability to deliver read, write, and execute commands to the FEP via a mailbox. With these commands the BEP may complete loading of the executable code into the FEP I-cache and begin execution of that Front End Processors stand alone code.

The procedures used to respond to the BEP directives are: `startUpFep`, `bootServerFep` and a (separate) copy of the FEP IO Library. No other code is active. When execution of a science run is commanded, the thread of control is passed to the function indicated and this boot code is effectively abandoned.

Note: This mechanism is not restricted to initiating science processing. Any code may be copied to the FEP, any code may be overwritten, and any code loaded may be executed and that code may or may not return. Since the thread of control is passed to an executing function, the responsibility to handle the watchdog is also passed to that function.

40.2 Uses

The Boot FEP provides the following feature:

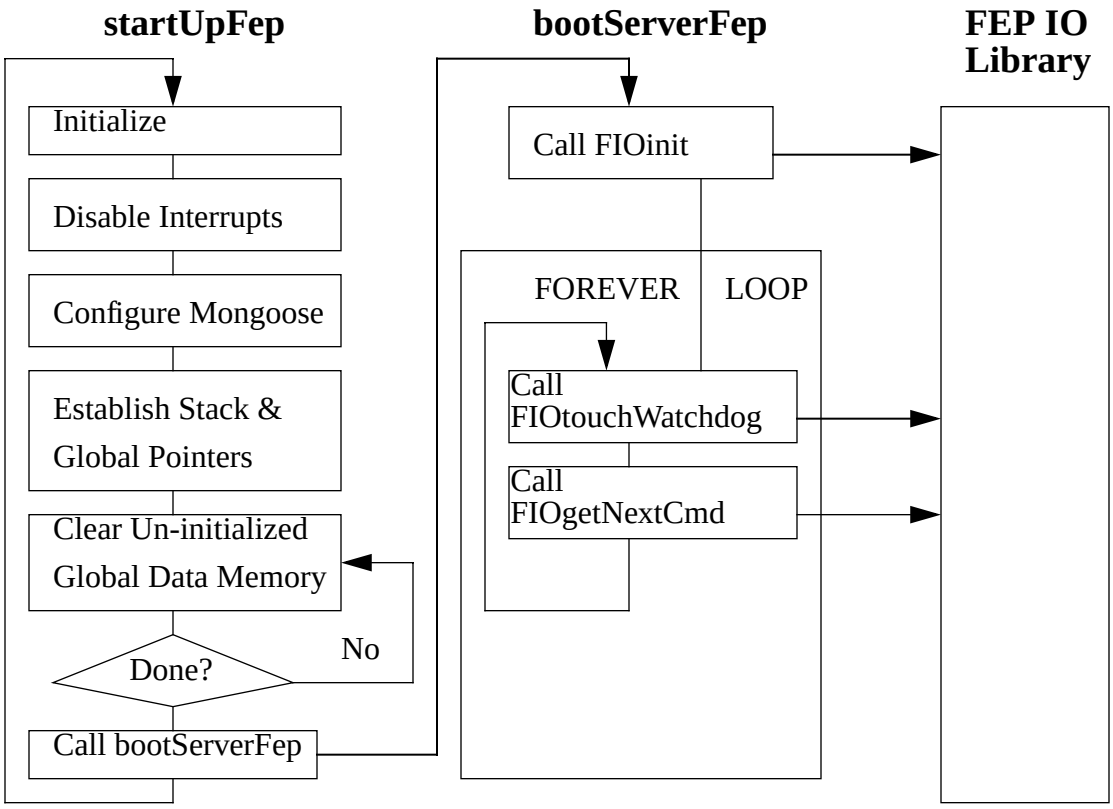
Use 1:: Provides a means of initiating executable code on the FEP.

40.3 Organization

Figure 187 illustrates the relationship between functions used by the FEP boot procedure.

1. The BEP/FEP interface is described in *DPA Hardware Specification & System Description* Rev. B section 2.1.2.10

FIGURE 187. FEP Boot Function Relationships



The FEP boot uses the FEP IO Library to fulfill commands entered into the incoming MailBox.

- The `startUpFep` routine establishes the mongoose hardware states needed to execute C code.
- The `bootServerFep` routine starts by calling `FIOinit` to set up the structures and pointers the FEP IO Library needs to provide communication from the BEP.
- It then loops FOREVER using `FIOtouchWatchdog` to reset the watchdog counter thus keep the watchdog from resetting the CPU, and using `FIOgetNextCmd` to respond to commands which will read, write, or to execute FEP memory.

40.3.1 Memory Map Requirements

The hardware locations of the MicroBoot Word (0xBFFFFFFC), reset vector (0xBFC00000), and general exception vector (0xBFC00180) are specified in section **2.2.2.3.1** of ***DPA Hardware Specification & System Description***, **Table 10. FEP Memory Map** (Rev B). (The Micro-Boot Word is aliased to 0xA017FFFC.) The linker must be constrained to establish the beginning of this boot code at the reset vector. When the reset line is released, execution will begin at the reset address.

With interrupts disabled, should a hardware exception be detected, control will pass to the exception vector. Code has been provided to copy the contents of relevant registers to shared memory. The client process may interrogate it to discern and record the cause of failure.

The MicroBoot word contains a “sequence to initialize critical configuration parameters that are not accessible under software control.”[1] Located at 0xBFFFFFFC, it is initiated upon reset.

40.4 Scenario

40.4.1 Pre-Boot Tasks

The functions `startUpFep` and `bootServerFep` are copied into FEP shared memory by the BEP while holding the FEP in a reset state. When released, the FEP will reference its microboot address and begin executing the `startUpFep` code.

40.4.2 Boot the FEP

`startUpFep` will initialize registers: setting kernel mode, disabling interrupts, mapping the general exception vector, setting co-processor 0 to usable, clearing the interrupt mask, and clearing the software interrupts in co-processor 0. (There are no additional co-processors.) The mongoose is then configured with no DMA and running with a single wait state. Because the stack size is declared during initial compilation, the stack pointer address is located at the maximum offset (minus this function’s save area of 24 bytes). The global data pointer (gp) is assigned to the linker defined symbol _gp.

C program specifications expect the global data area (.bss) to be cleared. The address of the first word and the end have been provided by the linker. The process enters a loop which will write zeros into those memory locations. The process then calls the monitor function `bootServerFep` to service commands from the BEP. Should that function ever return, `startUpFep` will continue execution from its first statement.

In the event that an anomalous condition is encountered which transfers control to the exception vector; `startUpFep` will copy the FEP CPU cause register, and the Coprocessor status, cause, and exception program counter registers to FAULT_DATA in shared memory. The watchdog will be set to expire quickly.

40.4.3 Monitor the Watchdog and Fulfill BEP Commands

`bootServerFep` establishes a buffer for commands and for the FIO`getNextCmd` status variable. It first calls `FIOinit` to initialize the FEP IO Library pointers and structures used in communicating commands from the BEP mail box. Next it enters a forever loop in which it calls `FIOtouchWatchdog` to keep the mongoose watchdog counter from reaching zero and resetting

1. Mongoose ASIC Microcontroler Programming Guide; 8.5 MSF Microboot: Brian S. Smith

the CPU and calls `FIOgetNextCmd` to have the library functions interrogate the BEP mailbox for any read, write, or execute in memory commands, to which other library functions will respond and fulfill. Commands other than those mentioned are not honored. Usually the BEP uses these functions to complete loading of memory and to begin executing a loaded function. Refer to FEP IO Library Section 39.0 for a more detailed description of the library functions.

40.5 Specification

40.5.1 bootServerFep()

Scope: Initialization

Return Type: void

Documentation:

This function will use the FEP IO Library functions to establish communications from the BEP and to respond to BEP read, write, or execute in memory commands.

Preconditions:

The FEP mongoose and its co-processor must have been initialized and booted.

Semantics:

This function will use the FEP IO Library function `FIOinit` to establish mailbox communications from the BEP. It then enters a FOREVER loop in which it calls `FIOtouchWatchdog` to prevent the watchdog from counting down and resetting the CPU, and calls `FIOgetNextCmd` to respond to BEP commands to read, write, or to execute in memory.

Other BEP commands are ignored. They will not generate a mailbox response.

Postconditions:

This function never returns.

40.5.2 startUpFep()

Scope: Boot

Return Type: void

Documentation:

This assembly language function establishes the necessary initial conditions on the FEP.

Semantics:

This function initializes registers, loads the mongoose configuration, initializes the stack and global pointers, clears the global data memory (.bss), and transfers control to `bootServerFEP`. Should that function return, `startUpFep` will restart.

In the event that an anomalous condition is encountered which transfers control to the exception vector; `startUpFep` will copy some significant registers to FAULT_DATA in shared memory. The watchdog will be primed to expire.

Postconditions:

This function never returns.

41.0 FEP Command Controller (36-53236 A)

41.1 Purpose

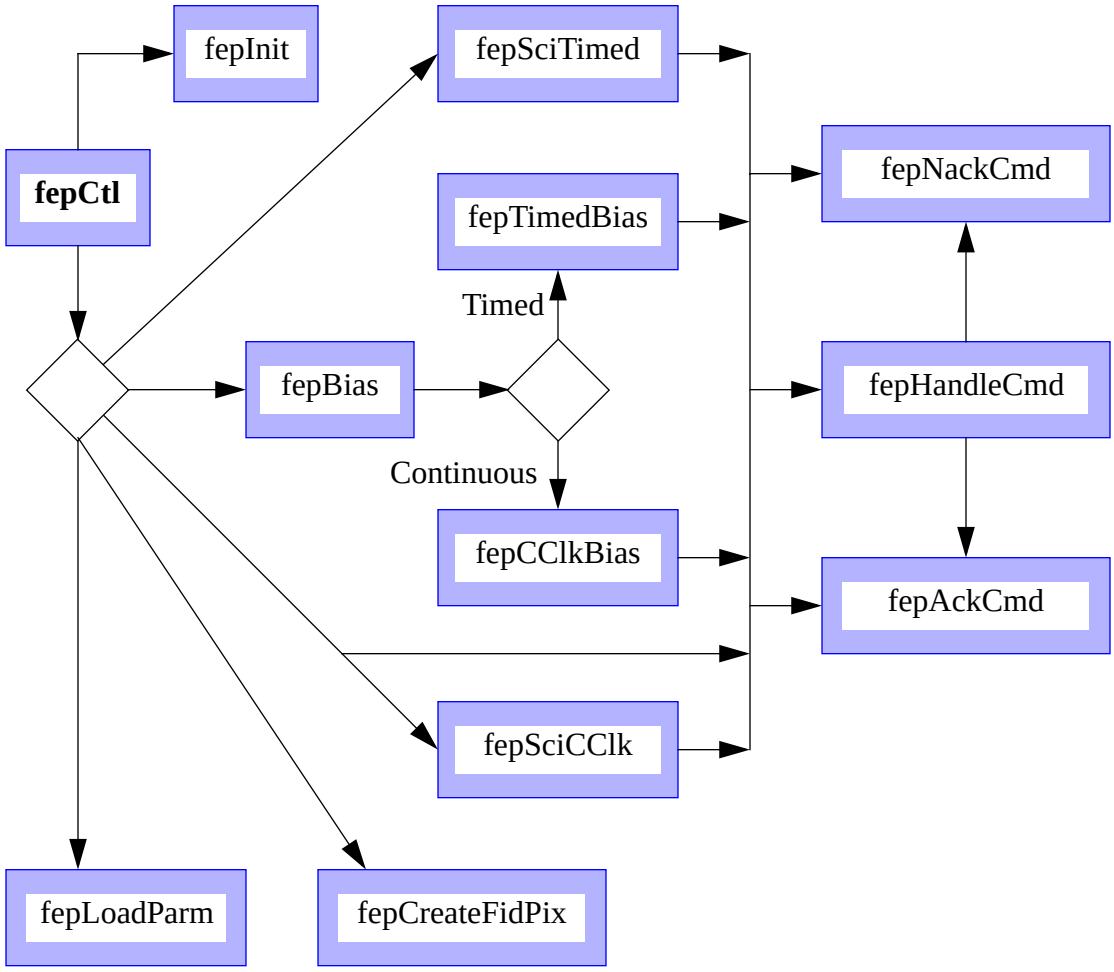
`fepCtl` performs high-level control functions within the FEPs, namely the interception of science commands from the BEP and their subsequent execution. Low level commands (memory read, write, and execute) are handled within the **FEP IO Library (36-53223 B)** routines (see Section 39.0) and are essentially transparent to the command controller and science modules.

41.2 Uses

The FEP Controller provides the following features:

Use 1:: Respond to a BEP science command from IDLE status
Use 2:: Respond to one BEP command while executing another

FIGURE 188. FEP controller subroutines and their calling hierarchy



41.3 Organization

The FEP Controller module contains the main FEP command processing loop, along with various external functions for frequently used FEP operations. Its stack contains a single copy of the `FEPparm` structure, whose address is passed to all FEP modules and functions. Thus if, for instance, `fepCtl` calls `fepSciTimed` to conduct a timed exposure science run, and `fepSciTimed` itself calls `fepHandleCmd` to process an incoming BEP command, it is `fepSciTimed`'s responsibility to ensure that it passes to `fepHandleCmd` the same pointer to `FEPparm` that it was itself given.

A brief description of the functions within this module is as follows:

- `fepCtl` - This is the top-level FEP processing loop. It allocates a single `FEPparm` structure on its stack and passes its address to all lower level science routines which can therefore use it as a substitute for `static` storage. By convention, this address parameter is called *`fp`* in all FEP modules. After calling `FIOinit` and `fepInit`, the code cycles endlessly over calls to `FIOgetNextCmd`. A `TRUE` return value signifies that a science-mode command has been received from the BEP, and the appropriate subroutine is called. The BEP itself will wait for the FEP to reply before sending another command. Finally, whether or not a command is processed, a call to `FIOtouchWatchdog` resets the watchdog timer, and the loop repeats.
- `fepAckCmd` - This function is called to respond positively to a BEP science command. It is global, since it can be called from other FEP science modules.
- `fepAppendRingBuf` - This function is called to add a block of data to the ring buffer, from whence it will be read by the BEP. The routine is global since it is only called from other FEP science modules.
- `fepBias` - This function is called when a bias calibration command is received from the BEP. It inspects the `FEPbiasRec` *`type`* code in *`fp->tp`*, and calls the appropriate bias function.
- `fepCreateFidPix` - This function is called when a `BEP_FEP_CMD_FIDPIX` command is received from the FEP. It removes any previously defined fiducial pixels, saves the addresses of new ones in *`fp->fidpix`* and purposely sets the corresponding bits of the bias parity plane to incorrect values, *i.e.* 0 becomes 1 and 1 becomes 0, in order to force the FEP to examine these pixels after every exposure.
- `fepEnableNextFrame` - This function is called to tell the FEP hardware thresholder to start storing image pixels when it next receives a `VSYNC` signal. It is global since it is only called from other FEP science modules.
- `fepHandleCmd` - This function is called within the `fepCtl` loop, and may also be called from any science function, when a positive return code from a call to `FIOgetNextCmd` indicates that the BEP has issued another science command.
- `fepInit` - This function is called once at start-up time. It initializes the *`flags`* field and the `FEPbiasRec`, `FEPstatus`, `FEPexpRec`, and `FEPexpEndRec` structures within the `FEPparm` structure.

- `fepLoadParm` - This function is called when a parameter block is received from the BEP via the BEP-FEP command mailbox. It copies the block to *`fp->tp`*, performs a series of validation tests, and then calls either `fepAckCmd` (or `fepNackCmd)` to tell the BEP that the block was received and that it passed (or failed) the validation.
- `fepNackCmd` - This function is called to respond negatively to a BEP science command, or when the command code returned by the `FIOgetNextCmd` call is not recognized. It is global, since it can be called from other FEP science modules.
- `fepSetAddrMode` - This function is called when initializing the various command modes to set the FEP hardware registers according to the desired DEA output node configuration. The routine is global since it is only called from other FEP science modules.

41.4 Global Variables

Table 40 lists those FEPparm fields that are referenced within the FEP command controller module. They are defined in *fepCtl.h* and always addressed by a pointer parameter named *fp*:

TABLE 40. Global FEPparm fields used by the Command Controller

FEPparm Substructure	Variable Name	Description
bepCmd		Latest command received from BEP, and the FEP's reply
	args	Command/reply contents
	len	Command/reply length in fullwords
	type	Command/reply type code (see *fepBep.h*)
br		Bias calibration parameters
	bias0[4]	Average overclock values of each output node for the first data frame used for bias calibration
	biassum	Checksum of bias0 values, initialized to 0xffffffff unless valid bias parameters are extracted from I-cache
ex		FEPexpRec exposure record
	expnum	Current exposure frame
	timestamp	Current frame's arrival time
	type	Initialized to FEP_EXPOSURE_REC
exend		FEPexpEndRec end-of-exposure record
	type	Initialized to FEP_EXPOSURE_END_REC
fepStatus		FEP status reported to BEP
	biasflag	TRUE if a valid bias map exists, else FALSE
	mode	Initialized to zero
	parityplane	Address of the start of the bias parity plane
	bias0[4]	Initialized to *bias0[4]*, if bias valid, otherwise zeroes
flags		Flag bits defined in *fepCtl.h*
	FP_SUSPEND	BEP has sent BEP_FEP_CMD_SUSPEND
	FP_TERMINATE	BEP has sent BEP_FEP_CMD_STOP
tp		Exposure parameter block
	btype	Initialized to FEP_NO_BIAS (see *fepBep.h*)
	ncols	Number of pixels per output node
	quadcode	Output node clocking mode (see *fepBep.h*)
	type	Initialized to FEP_NO_PARM (see *fepBep.h*)
fidpix[MAX_FID_PIX]		Row and column addresses of fiducial pixels.
nfidpix		Number of fiducial pixels to be reported in timed-exposure event detecting modes, set by the most recent call to fepCreateFidPix.
nextexpnum		Next possible frame number to be processed, set by a call to fepEnableNextFrame

41.5 Scenarios

41.5.1 Use 1: Respond to a BEP science command from IDLE status

The `fepCtl` loop receives a positive return code from a call to `FIOgetNextCmd`, signifying that a command has been received from the BEP and is sitting in the BEP-to-FEP mailbox. The command *`type`* is copied to *`fp->fepStatus.mode`* and examined to determine the action to be taken, as shown in Table 41.

Parameter loads (BEP_FEP_CMD_PARAM) and status requests (BEP_FEP_CMD_STATUS) are handled entirely within the ***fepCtl*** module and have no immediate external consequence. Commands to start science runs (BEP_FEP_CMD_TIMED and BEP_FEP_CMD_CCLK) and bias calibrations (BEP_FEP_CMD_BIAS) cause the corresponding science modules to be invoked. The remaining commands—BEP_FEP_CMD_STOP to terminate a running command, and BEP_FEP_CMD_SUSPEND and BEP_FEP_CMD_RESUME to temporarily suspend and restart a command—have no meaning since no command is running. `fepCtl` therefore responds by calling `fepNackCmd` to indicate to the BEP that this was the case.

Note that if ***fepCtl*** calls another module to execute the command, it is the responsibility of the called function to make a prompt call to either `fepAckCmd` or `fepNackCmd` to tell the BEP whether the command was accepted or rejected.

TABLE 41. FEP responses to BEP commands received in IDLE mode

Value of fp->bepCmd.type*	Function Called	Description
BEP_FEP_CMD_BIAS	`fepBias`	Start a bias calibration
BEP_FEP_CMD_CCLK	`fepSciCClk`	Start a continuously clocked science run
BEP_FEP_CMD_FIDPIX	`fepCreateFidPix`	Load zero or more fiducial pixel addresses into *`fp->fidpix`*.
BEP_FEP_CMD_PARAM	`fepLoadParm`	Load and check a new FEP parameter block
BEP_FEP_CMD_RESUME	`fepNackCmd`	Signal the BEP that the command is inappropriate in IDLE mode
BEP_FEP_CMD_STATUS	`fepHandleCmd`	Return the current FEP software status
BEP_FEP_CMD_STOP	`fepNackCmd`	Signal the BEP that the command is inappropriate in IDLE mode
BEP_FEP_CMD_SUSPEND	`fepNackCmd`	Signal the BEP that the command is inappropriate in IDLE mode
BEP_FEP_CMD_TIMED	`fepSciTimed`	Start a timed exposure science run
None of the above	`fepNackCmd`	Signal the BEP that the command is not understood

* see the `#define` statements in *fepBep.h.*

41.5.2 Use 2: Respond to one BEP command while executing another

The “high-level” science processing routines—`fepSciTimed`, `fepSciCClk`, and their corresponding bias calculation routines, `fepTimedBias`, and `fepCClkBias`—should make frequent calls to `FIOgetNextCmd` to determine whether a command has been received from the BEP. When this function returns `TRUE`, it indicates that a high-level command has been received via the BEP-FEP mailbox.[1] The science routine should immediately call `fepHandleCmd`, which uses the value of *`fp->bepCmd.type`* to determine the action to take, as shown in Table 42:

TABLE 42. FEP responses to BEP commands received during a science or bias run

Value of fp->bepCmd.type	Function Called	Description
BEP_FEP_CMD_BIAS	`fepNackCmd`	This command is unanticipated while a science or bias run is in progress.
BEP_FEP_CMD_CCLK	`fepNackCmd`	This command is unanticipated while a science or bias run is in progress.
BEP_FEP_CMD_FIDPIX	`fepNackCmd`	This command is unanticipated while a science or bias run is in progress.
BEP_FEP_CMD_PARAM	`fepNackCmd`	This command is unanticipated while a science or bias run is in progress.
BEP_FEP_CMD_RESUME	`fepAckCmd`	Signal that processing is to be resumed by clearing the FP_SUSPEND bit in fp->flags.
BEP_FEP_CMD_STATUS	`FIOwriteCmdReply`	Call `FIOwriteCmdReply` directly to reply to the BEP, passing as argument the current *`fp->fepStatus`* structure.
BEP_FEP_CMD_STOP	`fepAckCmd`	Signal that the run is to be terminated by setting the FP_TERMINATE bit in fp->flags.
BEP_FEP_CMD_SUSPEND	`fepAckCmd`	Signal that processing is to be temporarily suspended by setting the FP_SUSPEND bit in *`fp->flags`*. The suspension is performed entirely by the science process.
BEP_FEP_CMD_TIMED	`fepNackCmd`	This command is unanticipated while a science or bias run is in progress.
Any other value	`fepNackCmd`	-

1. Low-level commands from the BEP are handled transparently within `FIOgetNextCmd` and are never reported to the high-level science layer.

41.6 Specification

41.6.1 fepCtl()

Scope: Science.

Return Type: void.

Arguments: none.

Description:

Once the `bootServerFep` loader has copied the FEP executable image to I-Cache memory, the BEP sends it a `CMD_EXECUTE_MEM` command to branch to the `fepCtl` entry point. This function starts out by calling `FIOinit` to initialize the low-level library functions, and `fepInit` to initialize fields in `fepCtl`'s automatic `FEPparm` structure, as shown in Section 41.4.

`fepCtl` then executes an endless loop, alternately calling `FIOgetNextCmd` to see whether a command has arrived from the BEP, and `FIOtouchWatchdog` to reset the watchdog timer. Whenever `FIOgetNextCmd` returns a positive value, `fepCtl` inspects the *`bepCmd.type`* code and calls the appropriate function (see Table 41).

`fepCtl` never returns. This is a *good thing*, since it is either called directly from the FEP loader executing in bulk memory, or through a small assembler-language stub, and the return path will almost certainly have been overwritten long since.

41.6.2 fepAckCmd()

Scope: Science.

Return Type: void.

Arguments:

*FEPparm *fp*

Description:

A command previously received from the FEP is positively acknowledged by (a) setting the reply type to the *fp->bepCmd.type* of the original command, (b) setting the reply length to 2, (c) setting the single reply value to TRUE, and (d) calling `FIOwriteCmdReply` to send the reply back to the BEP.

41.6.3 fepAppendRingBuf()

Scope: Science.

Return Type: void.

Arguments:.

*unsigned *ptr*

unsigned wordcnt

*FEPparm *fp*

Description:.

`fepAppendRingBuf` calls `FIOappendBlock` to write *`ptr`* to the ring buffer in segments of no more than 32 words each. If `FIOappendBlock` returns `FALSE`, it indicates that a command has been received from the BEP, and `FIOgetNextCmd` will be called to process it. If `FIOgetNextCmd` returns `TRUE`, this was a high-level command and `fepHandleCmd` is then called to give it further processing. Once any command is processed, `fepAppendRingBuf` resumes its calls to `FIOappendBlock`, not returning until all *`wordcnt`* words have been copied to the ring buffer.

41.6.4 fepBias()

Scope: Static.

Return Type: void.

Arguments:

*FEPparm *fp*

Description:

The *fp->tp.type* and *fp->tp.btype* fields are inspected and the appropriate bias calibration routine called, as shown in Table 43.

The bias routine, `fepTimedBias` or `fepCClkBias`, must immediately inspect the parameter block, *fp->tp*, for validity and respond `fepAckCmd` or `fepNackCmd`, as appropriate, so that the BEP can be assured that the bias command has been received.

TABLE 43. Selection of Bias Calibration Function

fp->tp.type value	fp->tp.btype value	Function called	Comments
FEP_TIMED_PARM_RAW FEP_TIMED_PARM_HIST FEP_TIMED_PARM_3x3 FEP_TIMED_PARM_5x5	FEP_BIAS_1 FEP_BIAS_2	`fepTimedBias`	Start a timed-exposure bias calibration
	FEP_NO_BIAS	`fepAckCmd`	No bias calibration required
	Other	`fepNackCmd`	Unexpected *btype* value
FEP_CCLK_PARM_RAW FEP_CCLK_PARM_1x3	FEP_BIAS_1 FEP_BIAS_2	`fepCClkBias`	Start a continuously clocked bias calibration
	FEP_NO_BIAS	`fepAckCmd`	No bias calibration required
	Other	`fepNackCmd`	Unexpected *btype* value
Any other value, including FEP_NO_PARM	Any	`fepNackCmd`	Unexpected *type* value

41.6.5 fepCreateFidPix()

Scope: Science.

Return Type: void.

Arguments:

*FEPparm *fp*

Description:

This is called when `fepCtl` receives a `BEP_FEP_CMD_FIDPIX` command while in *`IDLE`* mode. It checks the *`fp->bepCmd.len`* field for legality[1], returning `FEP_CMD_ERR_PARM_LEN` to the BEP if illegal. It then determines whether the bias map is valid (is *`fp->fepStatus.biasflag true`*?). If not, it returns `FEP_CMD_ERR_NO_BIAS` to the BEP.

If the tests succeed, any existing fiducial pixels are cleared out of the bias parity plane (see below) and the stored count *`fp->nfidpix`* is set to zero. Then the first *`fp->bepCmd.len`*-1 elements of *`fp->bepCmd.args`* are accepted as defining the new fiducial pixel list for subsequent time exposure science runs using the current bias map. Within each 32-bit element, the 12 low order bits (0-11) define the column[2] and bits 16-27 define the row. For each new fiducial pixel, the corresponding bias parity plane bit is set to an *incorrect* value, relative to its 12-bit value in the bias map, thereby causing the hardware thresholder to flag it as a parity error on every exposure frame. This in turn will allow the timed-exposure event-detection routine (`FEPtestEvenPixel`) to recognize and store it as a fiducial pixel.

When the fiducial pixel address list has been processed, `fepCreateFidPix` calls `fepAckCmd` to pass a `FEP_CMD_NOERR` return code to the BEP.

1. $1 \leq$ *`fp->bepCmd.len`* $\leq$ `MAX_FID_PIX`.

2. Since fiducial pixels are always reported in contiguous even/odd pairs, an odd column address will be decremented upon receipt.

41.6.6 fepEnableNextFrame()

Scope: Science.

Return Type: void.

Arguments:.

*FEPparm *fp*

Description:.

This function is called to signal the FEP hardware to store then next image frame, i.e. the pixels that follow the next `VSYNC` code. This is done with a call to `fioWriteImPulseReg(IPULSE_ARMNXTACQ)`, sandwiched between calls to `FIOgetExpInfo`. If the exposure number is found to have changed between these calls, it implies that a VSYNC code was encountered. Since the software has no way of knowing whether the VSYNC was received before or after the pulse register was updated, the call to `fioWriteImPulseReg` will be issued a second time, causing a frame to be skipped.

On exit, `fp->ex.expnum` contains the current value of the exposure counter, `fp->ex.timestamp` contains the corresponding latched clock time, and `fp->nextexpnum = fp->ex.expnum + 1`.

41.6.7 fepHandleCmd()

Scope: Science.

Return Type: void.

Arguments:

*FEPparm *fp*

Description:

This function is called whenever a call to `fepGetNextCmd` returns `TRUE`, indicating that a BEP command has been received. The responses are determined by the *`bepCmd.type`* value and are listed in Table 42 on page 1211. In each case, a reply is sent to the BEP: usually either positive acknowledgment (`fepAckCmd`) or negative acknowledgment (`fepNackCmd`), or, in response to a BEP_FEP_CMD_STATUS request, `FIOwriteCmdReply` is called to return the contents of *`fp->fepStatus`* in the reply argument list.

41.6.8 fepInit()

Scope: Static.

Return Type: void.

Arguments: none.

Description:

This function is called to initialize the Front End Processor's `FEPparm` parameter block, as shown in Table 44. Bias values are restored from I-cache by a call to `FIOgetBiasConfig`. If `br.biassum` is consistent with the `bias0` array, it is assumed that the bias map itself is still usable, so `fepStatus.biasflag` is set `TRUE`. Otherwise, it is set `FALSE` and `br.biassum` is initialized to the "impossible" value `0xffffffff`.

TABLE 44. FEPparm variables Initialized by fepInit

Variable	Initial Value
br.biassum	`0xffffffff` unless the value copied from I-cache was consistent with the values of *br.bias0*`[]` also copied from I-cache.
ex.type	`FEP_EXPOSURE_REC`
exend.type	`FEP_EXPOSURE_END_REC`
fepStatus.mode	`0`
fepStatus.biasflag	`TRUE` if *br.biassum* is valid, otherwise `FALSE.`
fepStatus.parityplane	`FIOgetBiasParityPlanePtr()` value
fepStatus.bias0`[]`	*br.bias0*`[]` if *br.biassum* is valid, otherwise zeroes.
flags	`0`
nfidpix	`0`
tp.type	`FEP_NO_PARM`

41.6.9 fepLoadParm()

Scope: Static.

Return Type: void.

Arguments:

*FEPparm *fp*

Description:

This function copies a FEP parameter block from the command mailbox *fp->bepCmd.args* to *fp->tp*. It checks the parameters shown in Table 45 and calls `fepAckCmd` if they are valid, or `fepNackCmd` if they are not.

TABLE 45. FEPparm variables that are tested by fepLoadParm

Variable Name	Validity Test
btype	Any legal `fepBiasType` value
ncols	Even and within the range 2-256
noclk	Even and within the range 0-32
nrows	Within the range 1-1024
quadcode	legal `fepQuadCode` value
type	legal `fepParmType` value

41.6.10 fepNackCmd()

Scope: Science.

Return Type: void.

Arguments:

*FEPparm *fp*

fepCmdRetCode errno

Description:

A command previously received from the FEP is negatively acknowledged by (a) setting the reply type to the *fp->bepCmd.type* of the original command, (b) setting the reply length to 2, (c) setting the single reply value to *errno*, and (d) calling FIOwriteCmdReply to send the reply back to the BEP. *errno* values are tabulated in *fepBep.h*.

41.6.11 fepSetAddrMode()

Scope: Science.

Return Type: void.

Arguments:.

fepQuadCode quadcode

unsigned ncols

bool tHold

Description:

This function loads FEP hardware registers with values derived from *quadcode*, the current DEA clocking mode, and *ncols*, the number of pixels per output node per row, as shown in Table 46. It calls `fioClearBitCtrlReg` and `fioSetBitCtrlReg` to set the control register, and `FIOsetOffsetReg` to set the four address offset registers.

FEP science and bias modules call the `FIOsetAddrMode` function to set FEP hardware registers so that the FEP addressing hardware can correctly interpret the incoming image pixels. It does this on the basis of the *quadcode* and *ncols* parameters, the former to specify the DEA output node configuration, and the latter to show the number of pixels to be read from a CCD quadrant by each output node.

The *tHold* parameter determines whether FEP hardware thresholding and bias parity detection should be turned on (TRUE) or off (FALSE). Overclock processing is always enabled.

The result will be to initialize the FEP image control register, and the four address offset registers, as shown in Table 46. Note that the fourth (diagnostic) output clocking possibility, in which nodes A–D are clocked “backwards”, need not be distinguished from FEP_QUAD_ABCD within the FEP, and the latter is therefore used for both.

TABLE 46. FEP hardware register configuration for CCD output clocking modes

Value of quadcode	FEP Img Control Register	FEP Address Offset Registers			
		$Offset_0$	$Offset_1$	$Offset_2$	$Offset_3$
FEP_QUAD_ABCD	0x2a	0	2*ncols–1	2*ncols	4*ncols–1
FEP_QUAD_AC FEP_QUAD_BD	0x1a	0	2*ncols–1	2*ncols	4*ncols–1

42.0 FEP Timed Exposure Modes (36-53224 B)

42.1 Purpose

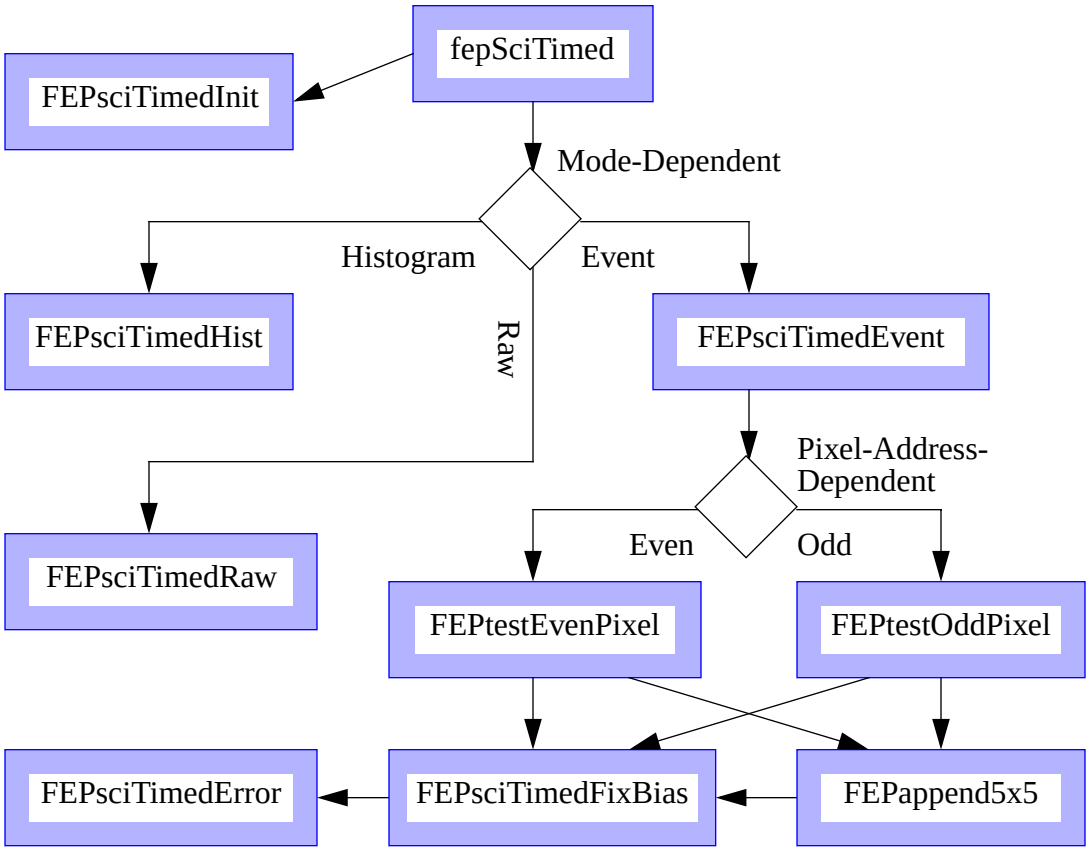
The ***fepSciTimed*** module, executing in the FEP, implements timed-exposure science processing, comprising either raw mode, event detection mode, or raw histogramming mode. It is called from `fepCtl` with a single argument, *`fp`*, a pointer to the `fepParm` structure. Before invoking `fepSciTimed`, the FEP must be commanded to (a) load a timed-exposure `FEPparmBlock` into *`fp->tp`*, and (b) perform a bias calibration.

42.2 Uses

The ***fepSciTimed*** function operates in one of the following modes, according to the value of *`fp->tp.type`*:

Use 1:: Identify and report 3x3 candidate X-ray events to the BEP.
Use 2:: Identify and report 5x5 candidate X-ray events to the BEP.
Use 3:: Report rows of raw pixels to the BEP.
Use 4:: Accumulate raw pixel histograms and report them to the BEP.

FIGURE 189. fepSciTimed subroutines and their calling hierarchy

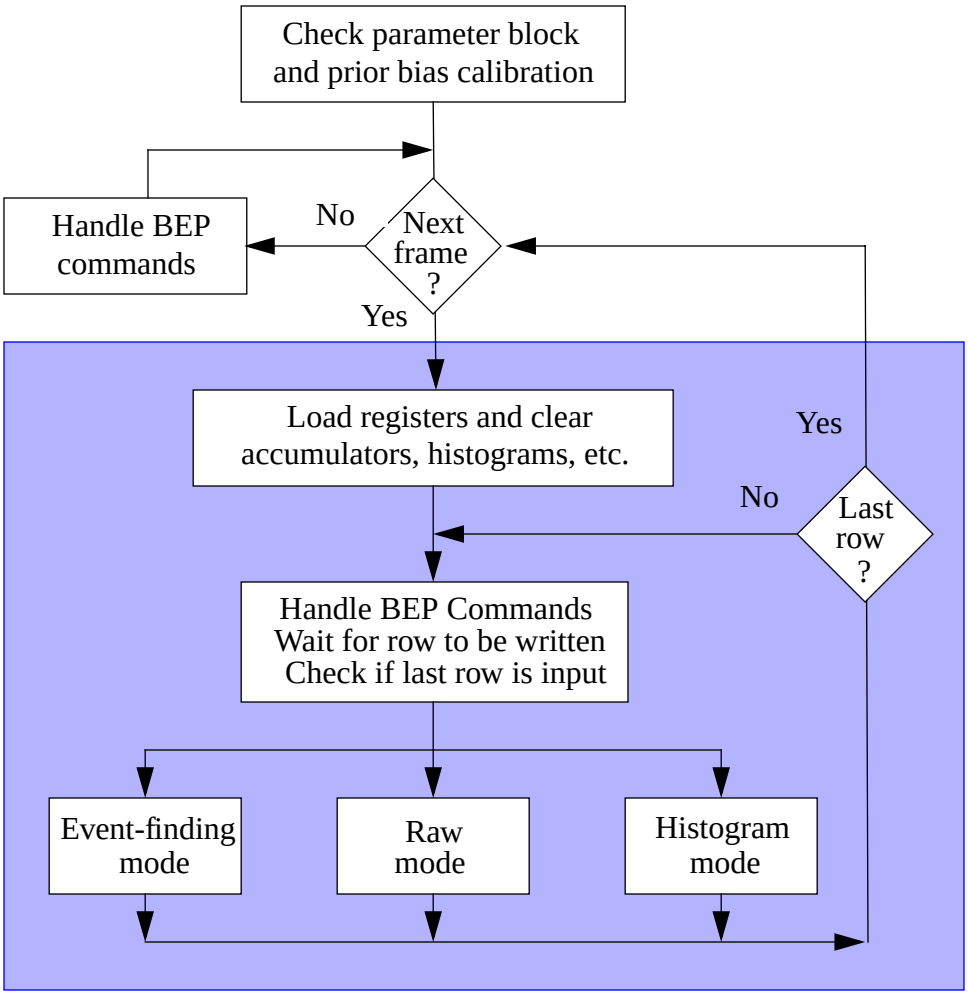


42.3 Organization

All interactions with the BEP and with FEP hardware are made through ***fepio*** library functions described in Section 39.0. The data structures that are passed between FEP and BEP are defined in that document and in Section 4.10. `fepSciTimed` makes use of the following functions which call each other in the manner shown in Figure 189:

- `FEPsciTimedInit`—validates the `FEPparmBlock`, *`fp->tp`*, and performs all necessary mode-dependent initialization, e.g. in event-finding modes, it checks that the bias map has been initialized.
- `FEPsciTimedEvent`—processes a single timed-exposure frame in event-finding mode, calling `FEPtestEvenPixel` and `FEPtestOddPixel` for each detected threshold crossing.
- `FEPsciTimedHist`—processes a single timed-exposure frame in raw histogram mode, reporting pixel frequency histograms and overclock statistics in `FEPeventRecHist` records.

FIGURE 190. The flowchart of fepSciTimed



- FEPsciTimedRaw—processes a single timed-exposure frame in raw mode, reporting raw pixels and overclocks in FEPeventRecRaw records without any thresholdingFEPtestEvenPixel—processes a threshold-crossing event possessing an even bit-offset relative to the start of the T-Plane buffer. If the center pixel is a local maximum, FIOappendData is called to copy a FEPeventRec3x3 record (or, in 5x5 mode, a FEPeventRec5x5 record) to the ring buffer.
- FEPtestOddPixel —processes a threshold-crossing event possessing an odd bit-offset relative to the start of the T-Plane buffer.If the center pixel is a local maximum, FIOappendData is called to copy a FEPeventRec3x3 record (or, in 5x5 mode, a FEPeventRec5x5 record) to the ring buffer.
- FEPsciPixTest—is an inline function (and therefore not shown in Figure 189) that is called several times within FEPtestEvenPixel and FEPtestOddPixel to determine whether the central pixel is less than (or equal to) one of its neighbors.
- FEPappend5x5—is called in 5x5 event-finding mode to (a) append the 16 edge pixels and bias values surrounding a core 3x3 event to a FEPeventRec3x3 structure before it is copied to the ring buffer by FEPtestEvenPixel or FEPtestOddPixel, and (b) change its type to FEP_EVENT_REC_5x5, thereby transforming it into a FEPeventRec5x5 structure.
- FEPsciTimedFixBias—is called if a parity error is discovered in one or both of a pair of bias map values. It calls FEPsciTimedError to reset the parity plane and T-plane bits, calls FIOappendData to copy a FEPerrorRec record to the ring buffer, and returns to the caller the corrected value of the bias map pair.
- FEPsciTimedError —resets the appropriate bits in the parity and threshold planes.

42.4 Global Variables

The following FEPparm fields, defined in *fepCtl.h* and invariably addressed by the fp pointer parameter, are used by all timed exposure modes:

bepCmd	latest command received from BEP
ex	current FEPexpRec record
exend	current FEPexpEndRec record
flags	flag bits:
FP_EDGE_ROW	current row is at the top or bottom of the frame
FP_SUSPEND	BEP has sent BEP_FEP_CMD_SUSPEND
FP_PAST_EOR	FEP hardware has finished with the current frame
FP_TERMINATE	BEP has sent BEP_FEP_CMD_STOP
nextexpnum	the next exposure index that the FEP is to process
quadrants	the number of DEA output nodes being sampled
tp	exposure parameter block
initskip	number of initial frames to ignore
ncols	number of CCD columns per output node
noclk	number of overclocks per output node
nrows	number of CCD rows between frame markers
quadcode	output node clocking mode
thresh[4]	threshold values for each output node
type	processing mode, either FEP_CCLK_PARM_1x3 or FEP_CCLK_PARM_RAW.

The following FEPparm fields are only used by fepSciTimedEvent*:*

br	pointer to bias calibration parameters
colshft	bit shift to transform column index to node index
fidpix[]	fiducial pixel address list
image	pointer to start of current image row
lastcol	index of last image column
nfidpix	fiducial pixel count

The following FEPparm fields are only used by fepSciTimedHist*:*

expcount	count of processed exposures
phist	pointer to FEPeventRecHist structure

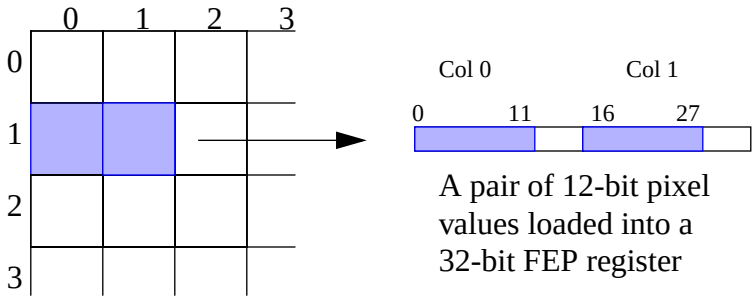
42.5 Scenarios

The following paragraphs describe the basic functions performed in timed exposure mode. All Timed Exposure modes wait until an image frame is processed and copied to the ring buffer before commanding the hardware thresholder to read the next frame, thereby ensuring that no partially processed frames are generated.

42.5.1 Use 1: Report 3x3 Events

- This mode is characterized by *fp->tp.type* == FEP_TIMED_PARM_3x3. After checking that the bias array contains appropriate values, FEPsciTimedInit calls fioWriteImPulseReg to start the hardware thresholder, and waits for frames to arrive. The first *fp->tp.initskip* frames are ignored.
- Once a valid frame arrives, fepSciTimed copies the exposure number and frame arrival time into a FEPexpRec structure and calls FIOappendData to copy it to the ring buffer. It then calls FEPsciTimedEvent to process the frame.
- FEPsciTimedEvent examines input rows one at a time. First, it calls FIOgetNextCmd to handle any incoming BEP commands, and then calls FIOgetExpInfo and/or FIOgetImageMapRowPtr until the hardware signals that the next row has been read into the image map.
- FEPsciTimedEvent then accumulates the current row's overclocks, adding them to the appropriate elements of *oSum*[], indexed by their DEA output nodes. It then inspects the T-plane buffer and calls either FEPtestEvenPixel or FEPtestOddPixel whenever a non-zero bit is found, indicating that the thresholder has located a pixel that exceeds its bias value by *fp->tp.thresh*[nn], where nn is the index of the appropriate DEA output node. Since the 12-bit pixel and bias values are only accessible two at a time via 32-bit CPU instructions, the logic required to inspect even-indexed pixels differs considerably from that needed for odd-indexed pixels, hence the two separate routines.

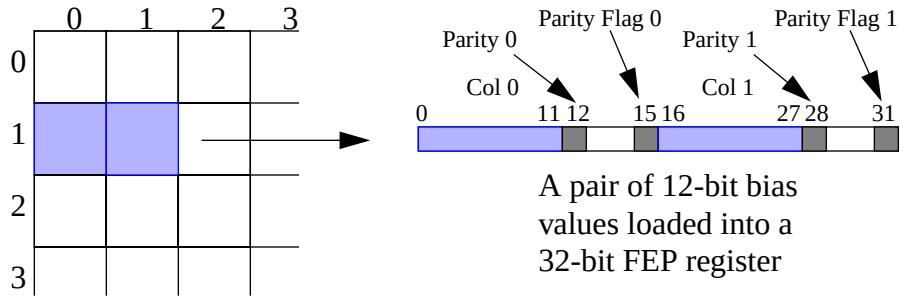
FIGURE 191. The Relation Between Image Pixels and FEP Register Values.



- The two pixel-testing routines examine the 8 pixels that surround the pixel that triggered the hardware thresholder, as illustrated in Figure 191. Pixels in the bad pixel list (with a bias value of 4095) are ignored, along with those whose bias values have been damaged since they were last calibrated, and were therefore set to 4094 during a sci-

ence run. If any of the remainder exceed the central pixel in relative (pixel–bias) value, or if any of the 4 pixels that follow the central pixel have the same relative value, the event is discarded; otherwise, `FIOappendData` is called to copy the 9 pixel and bias values to the ring buffer.

FIGURE 192. The Relation Between the Bias Map and FEP Register Values.



- Whenever `FEPtestEvenPixel` or `FEPtestOddPixel` loads a pair of bias values, as illustrated in Figure 192, it inspects their parity error flags (bits 15 and 31). If either is set, indicating that the parity of the bias value doesn't match the value of the corresponding bit in the bias parity plane, `FEPsciTimedFixBias` is called to reset the bias parity bit and the T-plane bit, and to return the "fixed" bias value (4094). The caller then stores the corrected value back into the bias map.
- Fiducial pixels are treated as a special case of bias parity errors, i.e. after the bias map and its parity plane are created, the BEP sends a `BEP_FEP_CMD_FIDPIX` command to the FEP, which calls the `fepCreateFidPix` routine (in ***fepCtl***). This sets the corresponding parity plane bits to *incorrect* values, which will cause the FEP thresholder to mark them in the T-plane. They will therefore always generate calls to `FEPtestEvenPixel` and it is the responsibility of `FEPsciTimedFixBias` to determine that they do not represent a genuine parity error, but to report them to the BEP by appending `FEPfidPixRec` records to the ring buffer.
- After processing the last row of pixels, `FEPsciTimedEvent` normalizes the *`oSum`*`[]` values and calls `FIOsetThresholdRegister` to update the hardware threshold registers.
- Finally, `FIOappendData` is called to copy a `FEPexpEndRec` (end-of-frame) record to the ring buffer.

42.5.2 Use 2: Report 5x5 Events

- This mode is characterized by *`fp->tp.type`* == `FEP_TIMED_PARM_5x5`. It is identical to the 3x3 mode described in the preceding section except that, prior to copying an event record to the ring buffer, `FEPtestEvenPixel` and `FEPtestOddPixel` call the `FEPappend5x5` function to transform the `FEPeventRec3x3` record into a `FEPeventRec5x5` record by setting its `type` to `FEP_EVENT_REC_5x5` and adding the 16 additional edge pixels and bias values.

42.5.3 Use 3: Report Raw Pixels

- This mode is characterized by *fp->tp.type* == FEP_TIMED_PARM_RAW. It begins in an identical manner to the event-finding modes described in the previous sections, except that fepSciTimed calls FEPsciTimedRaw to process each frame.
- FEPsciTimedRaw determines whether it can process the remainder of a current frame before it is overwritten by comparing the per-row processing time and data volume with the available time and ring-buffer space.
- FEPsciTimedRaw copies each row of pixels, and up to 30 overclocks from each DEA output node, to a FEPeventRecRaw record and thence to the ring buffer.

42.5.4 Use 4: Report Raw Pixel Histograms.

- This mode is characterized by *fp->tp.type* == FEP_TIMED_PARM_HIST. Processing commences in the manner described in the preceding sections, except that fepSciTimed calls FEPsciTimedHist to process each frame.
- FEPsciTimedHist adds each row's pixels to the histogram arrays, one per DEA output node.
- Each row's overclocks are also segregated by output node (*iquad*) and their cumulative minimum, maximum, sum, and sum-of-squares are saved, respectively, in the *iquad* elements of the *fp->fhist->omin*, *fp->fhist->omax*, *osum*, *ossqh*, and *ossql* arrays. Since the sum-of-squares may overflow an integer register, 64-bit arithmetic is used, i.e. the sum is stored in two 32-bit fields as (*ossqh*[*iquad*]+ BIGNUM**ossql*[*iquad*]), where BIGNUM, a constant defined in *fepCtl.h*, does not exceed 2^{32}-1.
- After processing each frame, the mean overclock value and its variance are computed for each output node. The mean value is simply the *osum* value divided by the number of overclocks summed, *nn*, i.e. in integer arithmetic, i.e., (*osum*[*nquad*]+*nn*/2)/*nn*. Computation of the variance is complicated by the need to preserve round-off accuracy. It is formally (*ossql*[*iquad*]+BIGNUM**ossqh*[*iquad*])/*nn*-(*osum*[*iquad*]/*nn*)*(*osum*[*iquad*]/*nn*). The calculation is broken down into the following steps:

  div = *osum*[*iquad*] / *nn*;

  rem = *osum*[*iquad*] % *nn*;

  variance = *ossqh*[*iquad*] * (BIGINT / *nn*) - *div* * *div*
  + (*ossqh*[*iquad*] * (BIGINT % *nn*) + *ossql*[*iquad*]
  - 2 * *div* * *rem* - (*rem* * *rem* + *nn*/2)/*nn* + *nn*/2)/*nn*;

- The mean overclock and variance values are then added to the *fp->fhist->omean* and *fp->fhist->ovar* arrays.
- After processing the last row of a series of *fp->tp.nhist* exposures, FEPsciTimedHist divides *fp->fhist->omean* and *fp->fhist->ovar* by *fp->tp.nhist*, and calls FIOappendData to copy the results, along with the pixel histograms themselves, to the ring buffer in a FEPeventRecHist structure.

42.6 Specification

This section describes the functions that are local to the `fepSciTimed` unit. The only external is `fepSciTimed` itself which is called from `fepCtl`. The `FEPparm` structure is defined in *fepCtl.h*, along with several pixel access macros. The interface to the FEP I/O library is described in Section 39.0, and data and messages exchanged between FEP and BEP are described in Section 4.10.

42.6.1 fepSciTimed()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

`FEPparm *fp`

Description:

`fepSciTimed` is called from `fepCtl` with a single argument: the pointer `fp` to the `fepParm` structure. It is responsible for all FEP actions associated with a timed exposure science run, except for bias calibration. It performs the following actions:

- Calls `FEPsciTimedInit` to validate the BEP's parameter block, *`fp->tp`*, for the current mode, to verify that pixel biases have been computed, and to initialize mode-dependent variables. If `FEPsciTimedInit` returns `FEP_CMD_NOERR`, `fepSciTimed` calls `fepAckCmd` and continues. Otherwise, it calls `fepNackCmd` to pass the error code to the BEP, indicating that the command has failed, and `fepSciTimed` then returns to `fepCtl`.
- `fepSciTimed` calls `FIOgetExpInfo` to determine the current exposure number and calls `fioWriteImPulseReg` to set the thresholder's `IPULSE_ARMNXTACQ` (write-enable) flag.
- `fepSciTimed` then enters a loop, labelled "next frame..." in Figure •, calling `FIOgetExpInfo` until the exposure numbered `fp->nextexpnum` is encountered (or exceeded). During this loop, calls are made to `FIOgetNextCmd` to handle any incoming BEP commands, and to `FIOtouchWatchdog` to prevent the Watchdog Timer from expiring while waiting for that particular frame.
- Once the desired exposure index is reached, i.e. once that frame starts to be processed by the FEP hardware, `fepSciTimed` writes a `FEPexpRec` record to the ring buffer and then calls either `FEPsciTimedEvent`, `FEPsciTimedRaw`, or `FEPsciTimedHist` to process the frame.

- `fepSciTimed` repeats the steps described in these last three bullets.
- The process continues until a `BEP_FEP_CMD_STOP` command is received from the BEP, which sets the `FP_TERMINATE` bit in *`fp->flags`*. When this occurs, `fepSciTimed` will finish processing the current frame before returning to `fepCtl`.

42.6.2 FEPsciTimedInit()

#include fepCtl.h

Scope: Science

Return type: fepCmdRetCode

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciTimed` to verify the contents of the `FEPparmBlock`, *`fp->tp`*, and to check whether a bias calibration has been performed. It also performs any necessary mode-dependent initialization.

If an error is detected, `FEPsciTimedInit` returns a *`fepCmdRetCode`* code as defined in *fepBep.h*; otherwise it returns `FEP_CMD_NOERR`.

42.6.3 FEPsciTimedEvent()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciTimed` to process a single timed exposure frame. It initializes the following pointers to the addresses of data structures used by the hardware thresholder:

pTPlane	points to T-plane buffer
pOclk	points to overclock buffer
fp->image	points to start of first data row

and these pointers will be advanced from row to row. The `FP_EDGE_ROW` bit in *fp->flags* is set to indicate that this is the first data row, and the four `oSum[]` overclock accumulators (one per DEA output node) are zeroed.

`FEPsciTimedEvent` then loops over CCD pixel rows. Once per row, it calls `FIOgetNextCmd` to see whether the BEP is trying to command the FEP. A returned value of `TRUE` signals that a science command has been received (utility commands will be executed within `FIOgetNextCmd` and will return `FALSE`), and a call will be made to `fepHandleCmd` to process it. NOTE: only one call is made to `FIOgetNextCmd` per incoming pixel row.

The function then processes the row data. It adds the overclocks to the `oSum` accumulators. Then it reads the T-plane, 32 bits at a time, until a non-zero value is found—indicating that the hardware detected a threshold crossing. Because the 12-bit pixel and bias values must be loaded in pairs, the 32 T-plane mask bits are tested two at a time—if an even-offset bit has been set, `FEPtestEvenPixel` is called; otherwise `FEPtestOddPixel` is called.

After processing the image row, `FEPsciTimedEvent` increments the *fp->image* pointer, and recomputes the `FP_EDGE_ROW` flag. After processing the last row, the *oSum* overclock accumulators are normalized and used to derive thresholds for the next exposure, which are communicated to the hardware by calls to `FIOsetThresholdRegister` for each DEA output node.

42.6.4 FEPtestEvenPixel()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from `FEPsciTimedEvent` for every bit set in the T-plane whose bit offset is even. This pixel is located in row *`irow`*, column *`icol`*, and *`fp->image`* points to the first pair of pixels in that row. The corresponding pair of bias values are to be found in *`fp->image`*`[BIAS_OFFSET]`.

FEPtestEvenPixel first loads the 32-bit pixel and bias fields that contain the central pixel. Since the FEP is a little-endian processor, and the central pixel is stored in the lower address pair of bytes, pixel and bias will be loaded into the least-significant 16 bits of each 32-bit variable. (The most significant 16 bits of these variables will contain the pixel and bias for row *`irow`* and column *`icol`*+1.)

The pair of bias values is now validated—the bias parity flags (bits 15 and 31) are tested to determine whether a parity error has occurred in either of those locations in the bias map. If so, a call is made to `FEPsciTimedFixBias` to handle the problem. Since this may instead be a fiducial pixel, the corrected 32-bit bias pair is examined; if it still contains a BIAS_PARITY flag, the central pixel is not fiducial—the pair of values is written back to the bias map, and the central pixel is ignored. Otherwise, processing continues.

Next, a test is made to determine whether this central pixel lies on the boundary of the CCD, or if its bias value is either BIAS_BAD (indicating that it had previously been found to have a parity error) or PIXEL_BAD (indication that the pixel is a member of the bad-pixel list). In any of these situations, the pixel is ignored.

The 12-bit value of the central pixel is saved in `ev.p[1][1]` and the corresponding 12-bit bias value in *`ev.b`*`[1][1]`. Two threshold values are loaded from the *`fp->ex.dOclk`*`[]` array—*`dql`* and *`dq`*—the former referring to the 3 pixels at a lower column number than the central pixel, the latter to the remaining 6 (which may belong to a different DEA output node). To assist in later `FEPsciPixTest`

calls, the variable *val* is set equal to the difference between the center pixel value and the sum of its bias and threshold values, as discussed in Section 42.6.6.

The center pixel is now compared to those lying to its left, to its right, and to those on the preceding row, using FEPsciPixTest, which also saves the pixel and bias values in the appropriate elements of the *ev.p* and *ev.b* arrays. If a test fails, i.e. if the central pixel isn't a local maximum, the function returns.

Before testing the row that follows the center pixel, FEPtestEvenPixel calls FIOgetExpInfo to see whether the hardware is still processing the same frame as the software. If so, it loops over calls to FIOgetImageMapRowPtr until the hardware has finished processing the current row. Then the three pixels in the following row are also tested against the center pixel using FEPsciPixTest. This time, however, it is also necessary to check each pair of bias values for a possible parity error. If discovered, FEPsciTimedFixBias is called to log the occurrence, and the fixed bias value pair is stored back in the bias map.

If all 8 pixels that surround the central pixel survive the FEPsciPixTest criteria, the 3x3 pixel and bias arrays in the *ev* structure will have been loaded. If 5x5 mode has been selected, FEPtestEvenPixel calls FEPappend5x5 to append the 16 bordering pixels and their corresponding bias values to the FEPeventRec3x3, transforming it into a FEPeventRec5x5 record. Finally, FEPtestEvenPixel calls FIOappendData to write the FEPeventRec3x3 record or FEPeventRec5x5 record to the FEP-BEP ring buffer.

42.6.5 FEPtestOddPixel()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from FEPsciTimedEvent for every bit set in the T-plane whose bit offset is odd. This pixel is located in row *irow*, column *icol*, and *fp->image* points to the first pair of pixels in that row. The corresponding pair of bias values are to be found in *fp->image*[BIAS_OFFSET]

FEPtestOddPixel first loads the 32-bit pixel and bias fields that contain the central pixel. Since the FEP is a little-endian processor, and the central pixel is stored in the upper address pair of bytes, pixel and bias will be loaded into the most -significant 16 bits of each 32-bit variable. (The least significant 16 bits of these variables will contain the pixel and bias for row *irow* and column *icol*–1.)

The pair of bias values is now validated—the bias parity flags (bits 15 and 31) are tested to determine whether a parity error has occurred in either of those locations in the bias map. If so, a call is made to FEPsciTimedFixBias to handle the problem, the corrected 32-bit bias pair is written back to the bias map, and the central pixel is ignored. Note that pixels from odd numbered columns cannot be fiducial (*cf.* FEPtestEvenPixel).

Next, a test is made to determine whether this central pixel lies on the boundary of the CCD, or if its bias value is either BIAS_BAD (indicating that it had previously been found to have a parity error) or PIXEL_BAD (indication that the pixel is a member of the bad-pixel list). In any of these situations, the pixel is ignored.

The 12-bit value of the central pixel is saved in *ev.p*[1][1] and the corresponding 12-bit bias value in *ev.b*[1][1]. Two threshold values are loaded from the *fp->ex.dOclk*[] array—*dqr* and *dq*—the former referring to the 3 pixels at a higher column number than the central pixel, the latter to the remaining 6 (which may belong to a different DEA output node). To assist in later FEPsciPixTest calls, the variable val is set equal to the difference between the center pixel value

and the sum of its bias and threshold values, as discussed in Section 42.6.6.

The center pixel is now compared to those lying to its left, to its right, and to those on the preceding row, using `FEPsciPixTest`, which also saves the pixel and bias values in the appropriate elements of the *`ev.p`* and *`ev.b`* arrays. If a test fails, i.e. if the central pixel isn't a local maximum, the function returns.

Before testing the row that follows the center pixel, `FEPtestOddPixel` calls `FIOgetExpInfo` to see whether the hardware is still processing the same frame as the software. If it is, it loops over calls to `FIOgetImageMapRowPtr` until the hardware has finished processing the current row. Then the three pixels in the following row are also tested against the center pixel using `FEPsciPixTest`. This time, however, it is also necessary to check each pair of bias values for a possible parity error. If discovered, `FEPsciTimedFixBias` is called to log the occurrence, and the fixed bias value pair is stored back in the bias map.

If all 8 pixels that surround the central pixel survive the `FEPsciPixTest` criteria, the 3x3 pixel and bias arrays in the *`ev`* structure will have been loaded. If 5x5 mode has been selected, `FEPtestOddPixel` calls `FEPappend5x5` to append the 16 bordering pixels and their corresponding bias values to the `FEPeventRec3x3`, transforming it into a `FEPeventRec5x5` record. Finally, `FEPtestOddPixel` calls `FIOappendData` to write the `FEPeventRec3x3` record or `FEPeventRec5x5` record to the FEP-BEP ring buffer.

42.6.6 FEPsciPixTest

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

```
FEPeventRec5x5 *ev

unsigned drow

unsigned dcol

unsigned val

unsigned pixel

unsigned code

unsigned bias
```

Description:

This is an `inline` function that is invoked several times within the `FEPtestEvenPixel` and `FEPtestOddPixel` functions. It masks the 12 low-order bits of `pixel` and `bias` and stores them in *`p`*`[`*`drow`*`][`*`dcol`*`]` and `b[`*`drow`*`][`*`dcol`*`]` in the `ev` structure. It then compares the value of (*`pixel - bias`*) against `val`, the corresponding value derived for the central pixel,

```
val = ev.p[1][1] - ev.b[1][1] - doclk;
```

where *`doclk`* represents an adjustment, quadrant by quadrant and frame by frame, for changes in average overclock. The averages from the first frame that was used in bias calibration are stored in *`fp->br.bias0`*`[]` and reported in `FEPexpRec` records, and all subsequent thresholds must be corrected by the difference between those *`bias0`* values and the current overclock averages. This difference is reported in the *`dOclk`* array in `FEPexpRec` records.

`FEPsciPixTest` evaluates to `TRUE` when the value of the pixel at (*`drow,dcol`*) is inconsistent with a legal event, or `FALSE` when it isn't. When the `code` argument is `TRUE` (non-zero), the comparison is "less-than-or-equal", i.e.

```
return val <= (pixel - bias);
```

This test is used for the 3 pixels whose drow index is greater than that of the central pixel, and for the pixel whose drow index is the same but whose *dcol* index is greater than the central pixel; for the remaining pixels, *code* should be set to FALSE (zero) and the comparison will be “less than”, i.e.

```
return val < (pixel - bias);
```

FEPsciPixTest also returns FALSE when the bias value is either PIXEL_BAD or BIAS_BAD, i.e. when a parity error has previously been detected in the *bias* value, or when the corresponding pixel is a member of the bad pixel list.

drow and *dcol* must be in the range 0-2; the “center” pixel has *drow*=*dcol*=1; *drow*=0 refers to the row before the center pixel, *drow*=2 to the row after; *dcol*=0 to the column before, *dcol*=2 to the column after. Constants PIXEL_MASK, PIXEL_BAD, and BIAS_BAD are defined within *fepCtl.h*, and *val* and *ev* are local variables.

NOTE: In 3x3 mode, only that part of the FEPeventRec5x5 structure which is identical to a FEPeventRec3x3 will be accessed by FEPtestEvenPixel and FEPtestOddPixel.

42.6.7 FEPappend5x5()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPeventRec5x5 *ev*

*unsigned *pImage*

*FEPparm *fp*

Description:

This function is called from `FEPtestEvenPixel` and from `FEPtestOddPixel` to copy the 16 edge pixels and their corresponding bias values to the *ev->pe*[] and *ev->be*[] arrays. In entry, *pImage* points to the 32-bit word in the pixel map that contains the central pixel of the event.

Since the calling routines will not have determined whether the last of the 5 rows has yet been processed by the FEP hardware, `FEPappend5x5` first calls `FIOgetExpInfo` to see whether the hardware is still processing the same frame as the software. If so, it loops over calls to `FIOgetImageMapRowPtr` until the 5th row (*ev->row*+2) is available.

The edge pixels and their bias values are then loaded, masked, and copied to *ev->pe*[] and *ev->be*[], taking care to detect and fix any bias parity violations via a call to `FEPsciTimedFixBias`.

42.6.8 FEPsciTimedFixBias()

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

unsigned bias

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from FEPtestEvenPixel or FEPtestOddPixel, (or FEPappend5x5*)* when a parity error or fiducial pixel is encountered in either or both of the halves of a 32-bit bias word.

FEPsciTimedFixBias first determines whether fiducial pixels have been defined (*fp->nfidpix* > 0), and also whether the parity flag is set for the even-column bias value. If both are true, the *fp->fidpix* array is scanned to see if it contains the current *irow*,*icol* address. If so, e.g. in *fp->fidpix*[*ii*], FEPsciTimedFixBias writes to the ring buffer a FEPfidPixRec record containing the index *ii* and the pair of pixel values, and it then removes the even pixel's parity error flag from the *bias* variable.

Fiducial pixels excepted, each parity-corrupted 12-bit value in *bias* is replaced by BIAS_BAD (defined in *fepCtl.h*) and FEPsciTimedError is called (twice if both 12-bit values are bad) to reset the appropriate bits in the Bias Parity Plane and T-plane. It then calls FIOappendData to send a message of type FEP_ERROR_REC to the BEP containing *irow*, *icol*, the uncorrected *bias* value, and the current exposure number in *fp->ex.expnum*. Finally, FEPsci-TimedFixBias returns the fixed-up 32-bit bias word to the caller, which has the responsibility for saving it back in the appropriate location in the bias map.

The returned value consists of the pair of input pixels and their parity error flags. If the even-column pixel was fiducial, the corresponding error flag is cleared in the returned value.

42.6.9 FEPsciTimedError()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from `FEPsciTimedFixBias` when a parity error is detected in row *irow*, column *icol* of the bias map. It resets the relevant bit in the Bias Parity Plane, turns off the corresponding bit in the T-plane, and increments the bias error count, *fp->exend.parityerrs*.

42.6.10 FEPsciTimedRaw()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciTimed` to process a single raw CCD image frame. It initializes the following pointers to the addresses of data structures used by the hardware thresholder:

pOclk	points to start of overclock buffer
image	points to start of first data row

and these pointers will be advanced from row to row.

`FEPsciTimedRaw` then loops over CCD pixel rows. Once per row, it calls `FIOgetNextCmd` to see whether the BEP is trying to command the FEP. A returned value of `TRUE` signals that a science command has been received (utility commands will be executed within `FIOgetNextCmd` and will return `FALSE`), and a call will be made to `fepHandleCmd` to process it. NOTE: only one call is made to `FIOgetNextCmd` per incoming pixel row.

The function then processes the row data, copying the raw pixels and overclocks to a `FEPeventRecRaw` structure and then calling `FIOappendData` to copy it to the ring buffer.

42.6.11 FEPsciTimedHist()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciTimed` to extract raw pixel histogram data from a single image frame. It initializes the following pointers to the addresses of data structures used by the hardware thresholder:

pOclk	points to start of overclock buffer
image	points to start of first data row

and these pointers will be advanced from row to row. If this is the first of a series of *fp->tp.nhist* frames, it saves the frame counter in the **fp->phist* histogram record and clears the histogram arrays.

`FEPsciTimedHist` then loops over CCD pixel rows. Once per row, it calls `FIOgetNextCmd` to see whether the BEP is trying to command the FEP. A returned value of `TRUE` signals that a science command has been received (utility commands will be executed within `FIOgetNextCmd` and will return `FALSE`), and a call will be made to `fepHandleCmd` to process it. NOTE: only one call is made to `FIOgetNextCmd` per incoming pixel row.

The routine accumulates each row of pixels into the *fp->phist->hist*[*nq*] histogram arrays, where *nq* represents the appropriate DEA output node index. It then examines the row's overclocks, updating the minimum (*fp->phist->omin*[*nq*]) and maximum (*fp->phist->omax*[*nq*]) values, and accumulating their sum (*osum*[*nq*]) and sum-of-squares. The latter are calculated in 64-bit arithmetic, using pairs of `unsigned` fields, *ossql*[*nq*] and *ossqh*[*nq*].

After each frame, mean overclocks and variances are summed into *fp->phist->omean* and *fp->phist->ovar*, as described in Section 42.5.4.

After processing the last line of the image frame, `FEPsciTimedHist` inspects *fp->ocount*[*nq*] and computes the mean overclock values (*fp->phist->omean*[*nq*]) and their variances (*fp->phist->ovar*[*nq*]) for each DEA output node, calling `FIOappendData` to copy the `FEPeventRecHist` record to the ring buffer.

43.0 FEP Timed Exposure Bias Calibration (36-53226 B)

43.1 Purpose

The ***fepTimedBias*** module, executing in the FEP, calibrates the bias map for subsequent timed-exposure science processing. It is called from ***fepCtl*** with a single argument, `fp`, a pointer to the `fepParm` structure. Before invoking ***fepTimedBias***, the FEP must be commanded to load a timed-exposure `FEPparmBlock` into *`fp->tp`*.

43.2 Uses

The ***fepTimedBias*** function operates in one of the following modes, according to the value of *`fp->tp.type`*:

Use 1:: Calculate bias using a “whole-frame” algorithm.
Use 2:: Calculate bias thresholds using a “strip” algorithm.

FIGURE 193. fepTimedBias Structure in “Whole-Frame” Mode

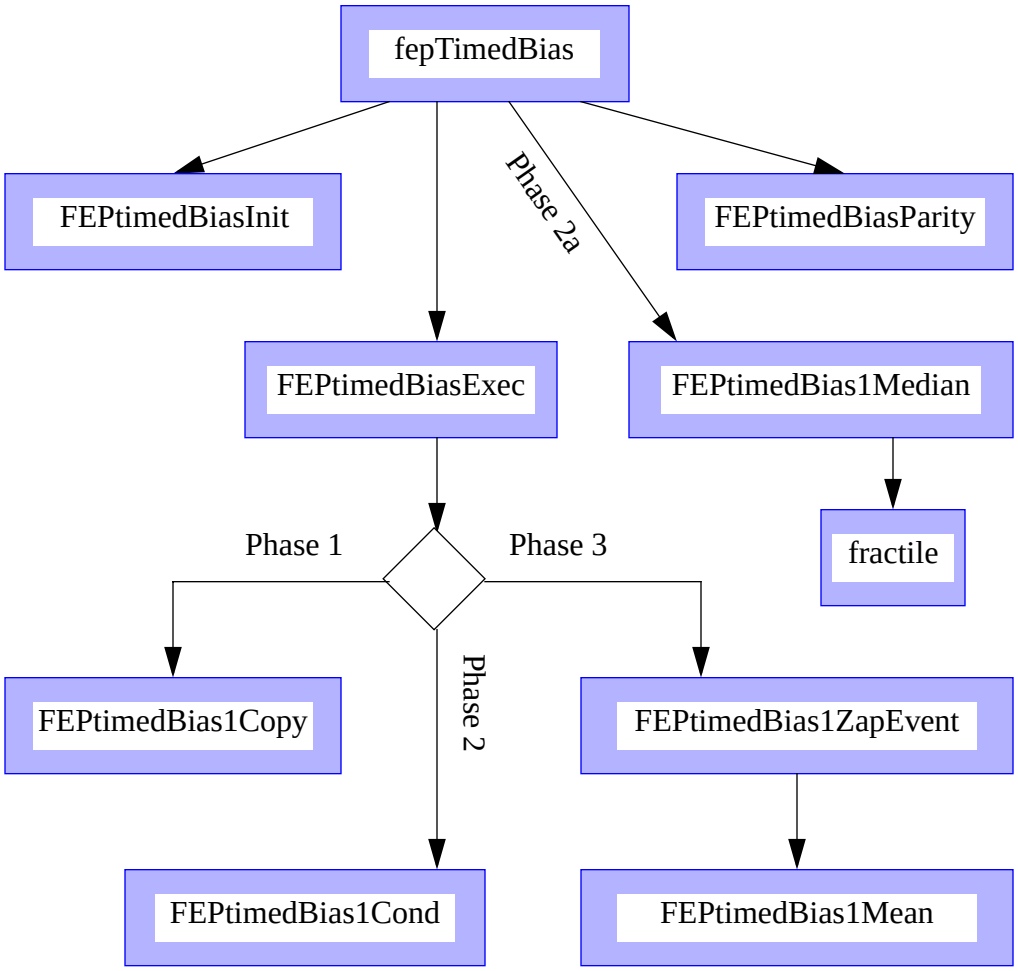
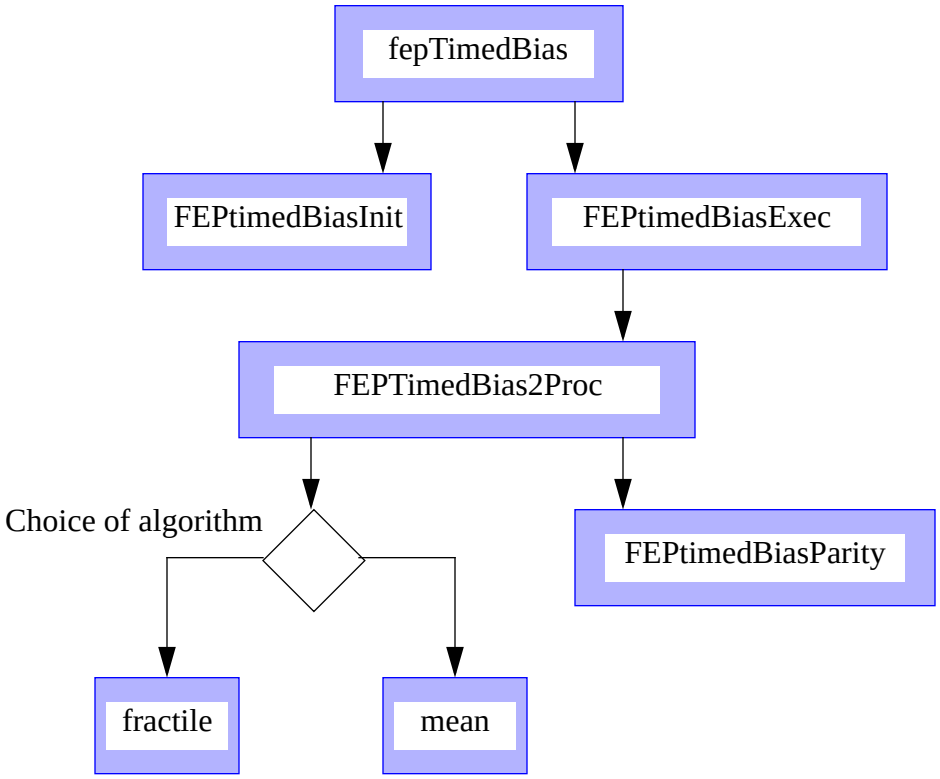


FIGURE 194. fepTimedBias Structure in "Strip" Mode



43.3 Organization

All interactions with the BEP and with FEP hardware are made through ***fepio*** library functions described in Section 39.0. The commands that are passed between FEP and BEP are defined in that document and in Section 4.10. ***fepTimedBias*** makes use of the following functions which call each other in the manner shown in Figure 193 ("whole-frame" mode) or in Figure 194 ("strip" mode), depending on the value of *`fp->tp.btype`* (see Table 47):

- ***FEPtimedBiasInit***—validates the `FEPparmBlock`, *`fp->tp`*, and performs all necessary mode-dependent initialization.
- ***FEPtimedBiasExec***—processes a single image frame. In "whole-frame" mode (Figure 193), it sums the overclocks and calls a subroutine, either ***FEPtimedBias1Copy***, ***FEPtimedBias1Cond***, ***FEPtimedBias1ZapEvent***, or ***FEPtimedBias1Mean***, depending on the value of the mode parameter, to process each line. In "strip mode" (Figure 194), this routine merely adjusts the hardware pointers for each strip until the image map is filled, when it sums the overclocks of the most recent exposure and calls ***FEPtimedBias2Proc*** to process the pixel values.
- ***FEPtimedBiasParity***—inspects one or more rows of bias map pixels and constructs a bias parity table in which each bit represents the parity (`EVEN=0`, `ODD=1`) of the corresponding bias value. In "whole-frame" mode, this routine is called once at the very end of the task. In "strip" mode, it is called after each strip of bias values has been created.

- ***FEPtimedBias1Copy***—copies a line of pixels from the image map to the bias map. This is called to process the first bias exposure in “whole-frame” mode.
- ***FEPtimedBias1Cond***—compares a line of pixels against a line of bias values, updating the bias with the corresponding pixel value when the latter is lower in value. This is called a total of *fp->tp.bparm*[0]-1 times in “whole-frame” mode.
- ***FEPtimedBias1ZapEvent***—compares each image pixel against its corresponding bias value. When the former exceed the latter by at least *fp->tp.bparm*[3], the image pixel and its 8 neighbors are set equal to PIXEL_BAD. This routine must be followed immediately by a call to ***FEPtimedBias1Mean***. The pair of routines are called a total of *fp->tp.bparm*[1] – *fp->tp.bparm*[0] – 1 times in “whole-frame” mode.
- ***FEPtimedBias1Mean***—examines each image pixel value, *p*, and, unless their value is PIXEL_BAD, replaces the corresponding bias value, *b*, by (*n***b*+*p*)/(*n*+1), where *n* is the post-conditioning exposure index, *fp->expnum - fp->tp.bparm*[0] + 1.
- ***FEPtimedBias1Median***—is only called in “whole-frame” mode (Figure 193). It will be called when the exposure number is *fp->tp.bparm*[0] and when *fp->tp.bparm*[2]is non-zero. It examines all 3×3 blocks of bias values. When the central value is less than all but one of its neighbors by at least *fp->tp.bparm*[2], it replaces the central value by the median of its neighbors. It should only be invoked when it is suspected that the image pixels contain anomalously low values that would otherwise corrupt the “whole-frame” bias map.
- ***FEPtimedBias2Proc***—is called from ***fepTimedBias*** once the image map contains a set of strips of pixels from multiple exposures of the same CCD rows. It extracts each set of pixels into a vector, calls ***mean*** or ***fractile*** to compute the bias value, and stores the result into the bias map. When the strips have been processed, it calls ***FEPtimedBiasParity*** to update the bias parity plane.
- ***mean*** is a utility function that returns the truncated mean of an array of values. It first computes the mean and RMS variance, then re-computes the mean, rejecting those values that differ from the first-order mean by more than a constant times the RMS variance. It is only called in “strip” mode.
- ***fractile*** is a utility function that sorts an array of values, and returns the value that is indexed by a constant, where the index is 0 for the smallest value, 1 for the next smallest, etc. Note that this function is used in both modes—in “whole-frame” mode, it is called by ***FEPtimedBias1Median*** to compute the median value of neighboring pixels; in “strip” mode, it is called from ***FEPtimedBias2Proc*** to compute the bias itself.

43.4 Global Variables

The following FEPparm fields, defined in *fepCtl.h* and *fepBep.h*, and is invariably addressed by the *fp* pointer parameter, are used by all timed exposure bias modes:

bepCmd	latest command received from BEP
br	pointer to bias calibration parameters
bias0[4]	average overclocks for first bias frame
biassum	sum of the 4 *bias0* values
ex	current FEPexpRec record
d0clk[4]	change in average overclock since last exposure
expnum	current exposure number
timestamp	microsecond timer value at start of expnum
expcount	number of bias frames processed
fepStatus	FEP status reported to BEP
biasflag	=1 if bias has been computed
flags	flag bits:

FP_SUSPEND	BEP has sent BEP_FEP_CMD_SUSPEND
FP_PAST_EOR	FEP hardware has finished with the current frame
FP_TERMINATE	BEP has sent BEP_FEP_CMD_STOP
FP_DONE	BEP is terminating normally

image	pointer to start of current image row
nextexpnum	the next exposure index that the FEP is to process
parity	pointer to 4096-element bias parity table
quadrants	the number of DEA output nodes being sampled
tp	exposure parameter block (see Table 47)
bparm[5]	mode-dependent parameters
btype	type of bias calibration desired
initskip	number of initial frames to ignore
ncols	number of CCD columns clocked
noclk	number of overclocks per node per row
nrows	number of CCD rows clocked
nskip	2-exposure alternation factor
quadcode	output node clocking mode

43.5 Scenarios

The following paragraphs describe the basic functions performed by ***fepTimedBias*** during timed exposure bias calibrations, which are determined by the fields in the parameter block, `fp->tp` shown in Table 47.

TABLE 47. Parameters used by fepTimedBias

Field Type	*Field Name*	*"Whole-Frame" Mode*	*"Strip" Mode*
unsigned	*nrows*	Number of bias rows to be calibrated.	
unsigned	*ncols*	Number of pixels per output node per row	
fepQuadCode	*quadcode*	Output node configuration, i.e., ABCD, AC, or BD.	
unsigned	*noclk*	Number of overclocks per row per output node	
fepBiasType	*btype*	FEP_BIAS_1	FEP_BIAS_2
int	*bparm*[0]	Number of conditioning exposures (PHASE2)	Number of exposures per pixel
int	*bparm*[1]	Number of approximation-to-mean exposures (PHASE3)	=0 to use ***mean*** =1 to use ***fractile***
int	*bparm*[2]	Rejection threshold for low-pixel elimination (immediately prior to PHASE3)	For ***mean***, specifies σ rejection criterion. For ***fractile***, index of sorted pixel array.
int	*bparm*[3]	Threshold for event rejection (PHASE3)	Ignored
int	*bparm*[4]	Rejection threshold for approximation-to-mean	Ignored
unsigned	*nskip*	Exposure skip factor; if non-zero, don't use those with "non-standard" exposure times for bias calibration.	

The bias calibration algorithms themselves are presented in some detail in the ACIS report entitled "*CCD Bias Level Determination*" by Rita Somigliana and Peter Ford, ACIS part #36–56012–02, MIT CSR, Revision 2, May 30, 1995.

43.5.1 Use 1: Calculate Bias using a Whole-Frame Algorithm

fepTimedBias calls ***FEPtimedBiasInit*** to check the `fp->tp` parameter block, to initialize the parity table (`fp->parity`), and set the hardware registers. It then loops, waiting for the next exposure to be received from the DEA. Subsequent processing occurs in three distinct phases, with an optional 2a phase invoked in special circumstances.

- *Phase 1: initialization*.—the image pixels p_i of the first exposure frame are copied directly to the bias buffer, forming the zeroth order bias values, b_i^0 .

$$b_i^0 = p_i \qquad \text{(EQ 1)}$$

- *Phase 2: conditioning*—a series of exposures are examined, *n*=1,*N*. In each, an image pixel will replace the corresponding bias pixel if the image pixel is lower in value.

$$\begin{aligned} b_i^n &= p_i && \text{if } p_i < b_i^{n-1} \\ &= b_i^{n-1} && \text{otherwise} \end{aligned} \qquad \text{(EQ 2)}$$

  After a number of exposures, typically 5–10 for the anticipated radiation levels, the chance of any bias map value being influenced by radiation is vanishingly small.[1] On the contrary, the values will typically be lower than the "true" bias values. If any of the pixel values in Phase 1 or 2 is anomalously low, e.g. more than 4σ lower than the mean, (σ is the standard deviation in measured values of that pixel), that value will become the bias value at the end of Phase 2, and must be filtered out by the optional Phase 2a before proceeding to Phase 3.

- *Phase 2a: fix-up by median filtering*—no new exposures are examined during this optional processing phase. Bias values that are much lower than their neighboring values are identified and corrected by median filtering, i.e.,

$$b_i^N = M[\{b_{i,\,i\pm1}^N\}] \qquad \text{if } b_i^N < \{b_{i,\,i\pm1}^N\} - BPARM[2] \qquad \text{(EQ 3)}$$

  where *M*[] represents the median of the 8 surrounding pixels.

- *Phase 3: approximation to the mean*—a further series of exposures, *m*=*N*+1,*M*, is examined. Each exposure is first examined for events, i.e. pixels that exceed their corresponding "conditioned" bias values by more than a threshold supplied in the parameter block. Once found, that pixel, and its immediate neighbors are set to a special "illegal" value.

$$p_{i\pm0,\,1}^m = 4095 \qquad \text{if } p_i^m > b_i^{m-1} + BPARM[3] \qquad \text{(EQ 4)}$$

  The same exposure is examined again. This time, only non-illegal pixels are considered. Those that do not exceed their bias values by more than a certain threshold are used to refine the bias level by a "running average" algorithm.

1. Care must be taken when observing a bright target during bias calibration that the pile-up in any single pixel doesn't violate this condition. At XRCF, bias calibration should probably be performed with the source turned off or with ACIS translated away from the focal axis.

$$b_i^m = \frac{m-N}{(m-N+1)} b_i^{m-1} + \frac{1}{(m-N+1)} p_i^m \quad \text{(EQ 5)}$$

if $p_i^m \neq 4095$ and $p_i^m < p_i^{m-1} + BPARM[4]$, and $b_i^m = b_i^{m-1}$ otherwise.

Before processing each line of image pixels, ***FIOgetNextCmd*** is called. If it returns TRUE, ***fepHandleCmd*** is called to process a single BEP command.

After the last exposure has been processed, ***FEPtimedBiasParity*** is called to construct a bias parity buffer that contains a single parity bit for each pixel in the bias map. This will be used by the FEP hardware to detect single-bit flips in the bias map during subsequent timed exposure science runs.

Finally, ***fepTimedBias*** exits with writes to the image map disabled. It returns to its command mode and waits for more BEP commands.

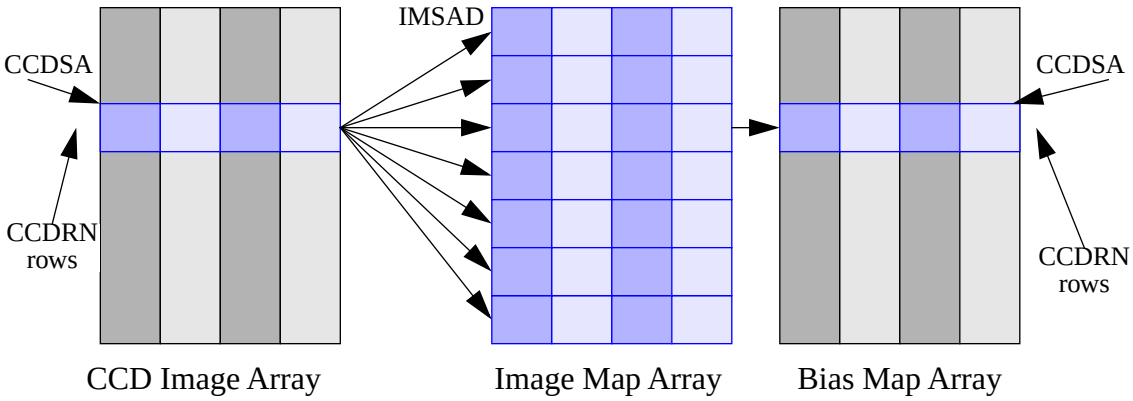
43.5.2 Use 2: Calculate Bias using a Strip Algorithm

fepTimedBias calls ***FEPtimedBiasInit*** to check the `fp->tp` parameter block, to initialize the parity table (`fp->parity`), and set the hardware registers. It then loops over calls to ***FIOgetExpInfo***, waiting for the next exposure to be received from the DEA.

The algorithms require each image pixel to be exposed several times, and its values after each exposure must be available simultaneously. Since there is insufficient memory available to store more than a single copy of the image map, the pixels must be processed a few at a time. On each exposure, the FEP hardware is therefore commanded to write only a strip of CCD image pixels to the image map. On subsequent exposures, the registers are adjusted so that each strip is written into a different part of the image map, as shown in Figure 195.

FIGURE 195. The Relation between CCD strips, Image strips, and exposures

A series of CCD exposures (left) is made and a single strip of CCDRN rows is copied to the image map (center), starting at row IMSAD. After each exposure, IMSAD is advanced by CCDRN. When the image map is full, it is processed and the resulting bias values are stored into the bias map (right), starting at row CCDSA. CCDSA is then advanced by CCDRN, IMSAD is reset to zero, and the process repeated for the next set of CCD strips. The register name mnemonics are taken from the ACIS SI Digital Processor Assembly, Hardware Specification and System Design, ACIS part #36–02104, Rev. A, MIT-CSR, October 4, 1995.



The choice of strip size is determined by the number of exposures desired (the value of `fp->tp.bparm[0]`), i.e. its size in rows is the smallest integer that does not exceed 1024 divided by the number of exposures. Once the image map is filled with strips, the program copies the corresponding pixel value in each strip to an array in D-cache for faster access. It then operates on the array values and uses either their truncated mean or their fractile as the bias value, which it stores in the bias map.

43.5.2.1 The Iterated Mean Algorithm

This algorithm, described in Section 3.2.4 of 36–56012–02, is implemented by the function ***mean*** (see Section 43.6.11). It takes the N pixel values p_i, i=0,N–1, and computes their mean value $\bar{p}$ and variance σ^2:

$$\bar{p} = \frac{1}{N}\sum_{i=0}^{N-1} p_i \qquad \text{(EQ 6)}$$

$$\sigma^2 = \frac{1}{N-1}\sum_{i=0}^{N-1} (p_i - \bar{p})^2 \qquad \text{(EQ 7)}$$

If the value of `BPARM[2]` is zero, ***mean*** returns $\bar{p}$ as the bias value. Otherwise, it inspects the p_i and removes any that do not satisfy the condition

$$|p_i| \le (\sigma \cdot \mathrm{BPARM}[2]) \qquad \text{(EQ 8)}$$

Finally, it recomputes $\bar{p}$ from the remaining p_i using Eq. 6, and returns that as the bias value.

43.5.2.2 The Fractile Algorithm

This algorithm, which is a generalization of the Median method described in Section 3.2.5 of 36–56012–02, is implemented in the function ***fractile*** (see Section 43.6.12). It takes the N pixel values p_i, i=0,N–1, sorts them into ascending order, and returns the value indexed by the value of `bparm[2]`. For instance, if N were 11 (it is usually 1024), and `bparm[2]` were 5, and the pixel values p_i were,

```
212 216 205 1041 208 217 211 214 215 206 210
```

their bias value would be 212, since, when the values are sorted into ascending order,

```
205 206 208 210 211 212 214 215 216 217 1041
```

that is the value of the element p_5.

43.6 Specification

This section describes the functions that are local to the ***fepTimedBias*** unit. The only external is ***fepTimedBias*** itself which is called from ***fepCtl***. The `FEPparm` structure is defined in *fepCtl.h* and *fepBep.h*, along with several pixel access macros. The interface to the FEP I/O library is described in Section 39.0, and data and messages exchanged between FEP and BEP are described in Section 4.10.

43.6.1 fepTimedBias()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

`FEPparm *fp`

Description:

fepTimedBias is called from ***fepCtl*** with a single argument: the pointer *`fp`* to the `fepParm` structure. It is responsible for all FEP actions associated with a timed exposure bias calibration. It performs the following actions:

- Copies the address of the *`par`* array in its D-cache stack, to *`fp->parity`*, where it will be initialized within ***FEPtimedBiasInit*** and used within ***FEPtimedBiasParity***.
- Calls ***FEPtimedBiasInit*** to initialize various `fepParm` fields and hardware registers. If ***FEPtimedBiasInit*** returns `FEP_CMD_NOERR`, ***fepTimedBias*** calls ***fepAckCmd*** and continues. Otherwise, it calls ***fepNackCmd*** to pass the error code to the BEP, indicating that the command has failed, and ***fepTimedBias*** then returns to ***fepCtl***
- Loops over exposures until either the `FP_DONE` or the `FP_TERMINATE` flag is set in *`fp->flags`*.
  - Entirely ignores the first *`fp->tp.initskip`* exposure frames.
  - When the bias type (*`fp->tp.btype`*) is “strip” mode (`FEP_BIAS_2`), sets the `mode` parameter to BIAS2. Otherwise, this is “whole-frame” mode and the `mode` parameter is set according to the number of fully-processed exposures (*`fp->expcount`*) and the *`fp->tp.bparm`* values. The first exposure will be processed as `BIAS1_PHASE1`, the next *`fp->tp.bparm`*[0] exposures will be processed as `BIAS1_PHASE2`, and the final *`fp->tp.bparm`*[1] exposures will be processed as `BIAS1_PHASE3`.

  - Calls ***fioWriteImPulseReg***, to set the `IPULSE_ARMNXTACQ` bit in the FEP's image pulse register.
  - Loops over calls to ***FIOgetExpInfo*** until the exposure number changes, i.e. until the hardware begins to write pixels from the next exposure into the image map. During this loop, calls are made to ***FIOgetNextCmd*** to intercept and process commands from the BEP, and ***FIOtouchWatchdog*** to keep the watchdog timer alive.
  - Calls ***FEPtimedBiasExec*** to process the exposure. NOTE: when the CCDs are clocked with two exposure times (i.e. non-zero *`fp->tp.nskip`*), ***FEPtimedBiasExec*** will not be called for those exposures with the initial (i.e. less frequent) exposure time.

- In "whole-frame" mode, calls ***FEPtimedBiasParity*** to initialize the bias parity buffer.
- Marks the bias map as "good" by setting *`fp->biasmode`* to `TRUE` and *`fp->br.biassum`* to the sum of the 4 elements in the *`fp->br.bias0`* array.

43.6.2 FEPtimedBiasExec()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

BiasMode mode

Description:

This function is called from ***fepTimedBias*** to process a single frame. *mode* selects either a phase of the "whole-frame" algorithm: BIAS1_PHASE1, BIAS1_PHASE2, or BIAS1_PHASE3, or the "strip" algorithm, BIAS2.

In the "whole-frame" algorithm, the routine loops over image frame rows, calling ***FIOgetNextCmd*** once per row to catch incoming commands from the BEP and, when detected, ***fepHandleCmd*** is called to process them. ***FEPtimedBiasExec*** then sums the overclocks and calls a subroutine to process the image pixels, depending on the *mode* value:

BIAS1_PHASE1—***FEPtimedBias1Copy*** copies a row of image pixels to the bias map without any change.

BIAS1_PHASE2—***FEPtimedBias1Cond*** reads a row of image pixels and uses them to replace the corresponding bias map values when the latter are larger than the former. The result of performing this operation over several consecutive exposures is to "condition" the bias map values, removing any contamination from CCD events.

BIAS1_PHASE3—first, ***FEPtimedBias1ZapEvent*** is called to mark all image map pixels that may contain events, followed by ***FEPtimedBias1Mean*** to use the unmarked pixels to update the bias values.

In the "strip" algorithm, when *mode* has the value FEP_BIAS_2, ***FEPtimedBiasExec*** does very little until the last of a set of strips has been written to the image map. It merely adjusts the hardware pointers via calls to ***FIOsetImageMapRowStart*** and ***FIOsetImageMapRowLength*** to march the strips down the image map, and to ***FIOsetCcdRowStart*** to select different rows of the CCD. After the image map is full of strips, ***FEPtimedBiasExec*** sums the overclocks of the last frame and then calls ***FEPtimedBias2Proc*** to update the bias map.

43.6.3 FEPtimedBiasInit()

#include fepCtl.h

Scope: Science

Return type: fepCmdRetCode

Arguments

*FEPparm *fp*

Description:

This function is called from ***fepTimedBias*** to verify the contents of the FEPparmBlock, *fp->tp*, and to perform the following initializations:

- *fp->quadrants* are set to 2 or 4, depending on the value of *fp->tp.quadcode*.
- *fp->parity*[0] *through fp->parity*[4095] are either set to PARITY_EVEN or to PARITY_ODD (defined in *fepCtl.h*) according to the bit parity of the binary integers 0–4095.
- FEP hardware registers are set by calls to ***FIOsetCcdRowStart***, ***FIOsetImageMapRowStart***, ***FIOsetImageMapRowLength***, and ***fepSetAddrMode***. Thresholding and bias parity error detection are disabled. Overclock processing is enabled.

If an error is detected, ***FEPtimedBiasInit*** returns a *fepCmdRetCode* value as defined in *fepBep.h*; otherwise it returns FEP_CMD_NOERR.

43.6.4 FEPtimedBiasParity()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

unsigned rowstart

unsigned nrows

Description:

In “strip” mode, this function is called from ***FEPtimedBias2Proc*** to update *nrows* in the bias parity buffer, starting at row *rowstart*, from the corresponding values in the bias map. In “full-frame mode”, it is called from ***FEPtimedBias*** to compute the entire parity buffer at the end of the bias calibration run. Each 12-bit bias map value is used as an index into the *fp->parity* array, whose elements are either PARITY_EVEN or PARITY_ODD, according to the parity of the index. For instance, the number fifteen is represented by the bit pattern 01111, which contains an even number of ‘1’s. Its parity is therefore even, so *fp->parity*[15] = PARITY_EVEN.

Precondition:

fp->parity must point to an array of 4096 32-bit values, which have been initialized to PARITY_EVEN if the index number possesses even parity, or to PARITY_ODD if odd.

43.6.5 FEPtimedBias1Copy()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***FEPtimedBiasExec*** in BIAS1_PHASE1 to copy a single row of pixels from the image map (located at *fp->image*) to the bias map (located at *fp->image* + BIAS_OFFSET).

Since this routine is only called for the first exposure of the bias calibration sequence, no overclock correction factor need be applied.

43.6.6 FEPtimedBias1Cond()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***FEPtimedBiasExec*** in BIAS1_PHASE2 to inspect a single row of pixels from the image map (located at *fp->image*) and update the corresponding values in the bias map (located at *fp->image* + BIAS_OFFSET) when the former are smaller than the latter.

Before making the comparison with the bias pixel, each image pixel is corrected for any change in average overclock by subtracting the *fp->ex.dOclk* element appropriate to the pixel's output node. Since the correction factor is based on the average overclocks from the previous exposure, this can only compensate for slow changes in the analog system.

43.6.7 FEPtimedBias1Mean()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***FEPtimedBiasExec*** in BIAS1_PHASE3 to read a single row of image pixels (located in *fp->image* - PIXEL_STRIDE) and update the corresponding bias values (located in *fp->image* + BIAS_OFFSET - PIXEL_STRIDE). Image pixels with the PIXEL_BAD value are skipped since they have been identified in a prior call to ***FEPtimedBias1ZapEvents*** as possibly containing events.

Before making the comparison with the bias pixel, each image pixel is corrected for any change in average overclock by subtracting the *fp->ex.dOclk* element appropriate to the pixel's output node. Since the correction factor is based on the average overclocks from the previously processed exposure, this can only compensate for slow changes in the analog system.

When the pixel value *p* exceeds the corresponding bias value *b* by not more than *fp->bparm*[4], *b* is replaced by (*n***b*+*p*)/(*n*+1), where *n* is the exposure count, i.e. *n*=1 for the first exposure of BIAS1_PHASE3, 2 for the second, etc.

Since ***FEPtimedBias1Mean*** is called immediately after ***FEPtimedBias1Cond***, and the latter is capable of nullifying pixels in the preceding row, ***FEPtimedBias1Mean*** must also work on that row, whose first pixel is located at *fp->image* - PIXEL_STRIDE, rather than on the current row, pointed to by *fp->image*.

43.6.8 FEPtimedBias1Median()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***fepTimedBias*** at the end of BIAS1_PHASE2 to identify anomalously low valued pixels in the bias map. It does this by comparing each bias value with those of its 8 neighbors. If the central value is smaller than all but one of its neighbors by more than *fp->tp.bparm*[2], ***FEPtimedBias1Median*** replaces the central value by the median of the neighbors.

This function is only invoked if *fp->tp.bparm*[2] is non-zero. It should only be used if it has been found from a study of previous bias maps that some anomalously low image pixel values will be encountered, since these would otherwise dominate the bias map that is constructed during BIAS1_PHASE2.

43.6.9 FEPtimedBias1ZapEvent()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

unsigned irow

Description:

This function is called from ***FEPtimedBiasExec*** in BIAS1_PHASE3 to identify image pixel values (at location *fp->image*) in row irow that are larger than their corresponding bias values (at location *fp->image* + BIAS_OFFSET) by more than a constant *fp->tp.bparm*[3].

Before making the comparison, each image pixel is corrected for any change in average overclock by subtracting the *fp->ex.dOclk* element appropriate to the pixel's output node. Since the correction factor is based on the average overclocks from the previous exposure, this can only compensate for slow changes in the analog system.

Once identified, the image pixels and their immediate neighbors (8 pixels, or less if the identified pixel is on the edge of the CCD) are reset to PIXEL_BAD so that they can be identified in the ***FEPtimedBias1Mean*** routine.

43.6.10 FEPtimedBias2Proc()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***FEPtimedBiasExec*** in the BIAS2 mode ("strip" processing) when the image map is filled with a series of strips of pixels from the same group of CCD rows from *fp->tp.bparm*[0] consecutive exposures.

Each image pixel is corrected for any change in average overclock by subtracting the *fp->ex.dOclk* element appropriate to the pixel's output node. Since the correction factor is based on the average overclocks from the last exposure that contributed to the current strip, this can only compensate for slow changes in the analog system.

Each set of pixel values, one from each exposure, is copied to a buffer in D-cache, *Data0* for pixels from even-indexed columns and *Data1* for pixels from odd-indexed columns. The value to be stored in the bias map is determined by the value of *fp->tp.bparm*[1].

bparm[1] = 0: ***mean*** is called to compute the mean of the pixel values. When fp->tp.bparm[2] is zero, this becomes the bias map value. When fp->tp.bparm[2] is non-zero, pixels with values that differ from the zeroth-order mean by more than fp->tp.bparm[2] times the RMS variance of the values are eliminated, and the mean of the remainder becomes the bias map value.

bparm[1] = 1: ***fractile*** is called to sort the pixel values into ascending order. The element in this sorted list indexed by fp->tp.bparm[2] becomes the bias map value.

After processing the entire set of strips in the image map, ***FEPtimedBias2Proc*** calls ***FEPtimedBiasParity*** to compute parity flags for each of the new bias map pixel values and store the result in the appropriate section of the bias parity buffer.

43.6.11 mean()

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

*unsigned *vec*

unsigned nvec

unsigned nsigma

Description:

This function returns the integer closest to the mean value of the *nvec*-element array *vec*[]. Half-integer values are rounded up. If *nsigma* is non-zero, the standard deviation (σ) of the elements is also calculated, and the mean is re-calculated from those elements of *vec*[] that differ from the original mean value by no more than *nsigma* $\times \sigma$. The algorithm is described in Section 43.5.2.1.

43.6.12 fractile()

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

*unsigned *vec*

unsigned nvec

unsigned nrep

Description:

This function sorts the elements of the *nvec*-element array vec[] into ascending value, and returns the value of the *nrep*'th element, i.e. *nrep*=0 returns the minimum, *nrep*=*nvec*/2 returns the median, etc. The routine uses Shell's method, which was chosen for its computational efficiency—*nvec* × ln(*nvec*)—and low start-up overhead. The algorithm is described in Section 43.5.2.2.

44.0 FEP Continuously Clocked Modes (36-53225 B)

44.1 Purpose

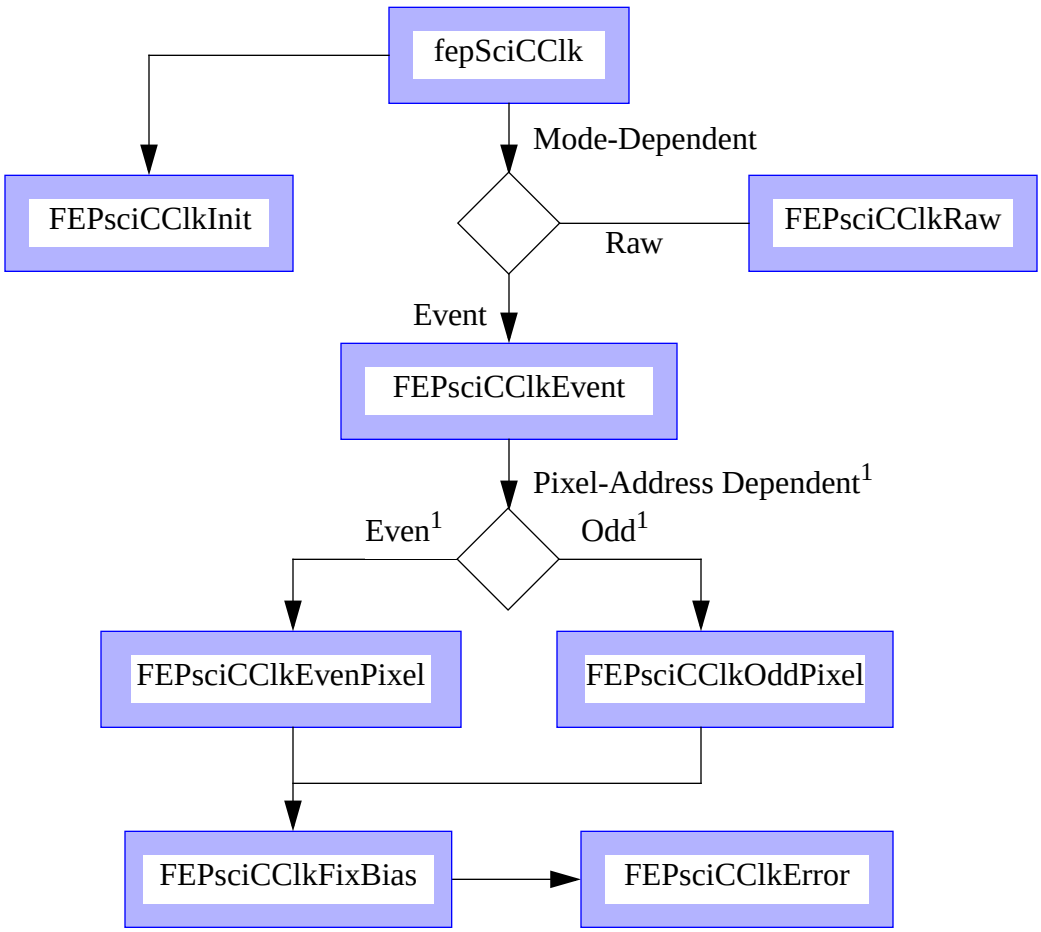
The `fepSciCClk` module, executing in the FEP, implements continuously clocked science processing, comprising either raw mode or event detection mode. It is called from `fepCtl` with a single argument, `fp`, a pointer to the `fepParm` structure. Before invoking `fepSciCClk`, the FEP must be commanded to (a) load a continuous clocking `FEPparmBlock` into `fp->tp`, and (b) perform a continuous clocking bias calibration.

44.2 Uses

The `fepSciCClk` function operates in one of the following modes, according to the value of *`fp->tp.type`*:

Use 1:: Identify and report 1x3 candidate X-ray events to the BEP
Use 2:: Report values of raw pixels and overclocks to the BEP

FIGURE 196. fepSciCClk subroutines and their calling hierarchy



44.3 Organization

All interactions with the BEP and with FEP hardware are made through `fepio` library functions described in Section 39.0. The data structures that are passed between FEP and BEP are defined in that document and in Section 4.10. `fepSciCClk` makes use of the following functions which call each other in the manner shown in Figure :

- `FEPsciCClkInit`—validates the `FEPparmBlock`, *`fp->tp`*, and performs all necessary mode-dependent initialization, e.g. in event-finding mode, it checks that the bias map has been initialized.
- `FEPsciCClkEvent`—processes a single continuously clocked frame (512 rows) in event-finding mode, calling `FEPsciCClkEvenPixel` and `FEPsciCClkOddPixel` for each detected threshold crossing. The flow of control is described in the shaded region of Figure 197.
- `FEPsciCClkRaw`—processes a single continuously clocked frame (512 rows) in raw mode, reporting raw pixels and overclocks in `FEPeventRecRaw` records without any thresholding, as shown in the shaded region of Figure 197.
- `FEPsciCClkEvenPixel`—processes a threshold-crossing event possessing an even bit-offset[1] relative to the start of the T-Plane buffer. If the center pixel is a local maximum, `fepAppendRingBuf` is called to copy a `FEPeventRec1x3` record to the ring buffer.
- `FEPsciCClkOddPixel` —processes a threshold-crossing event possessing an odd bit-offset[1] relative to the start of the T-Plane buffer.If the center pixel is a local maximum, `fepAppendRingBuf` is called to copy a `FEPeventRec1x3` record to the ring buffer.
- `FEPsciCClkPixTest`—is an `inline` function (and therefore not shown in Figure) that is called several times within `FEPsciCClkEvenPixel` and `FEPsciCClkOddPixel` to determine whether the central pixel is less than (or equal to) one of its neighbors.
- `FEPsciCClkFixBias`—is called if a parity error is discovered in one or both of a pair of bias map values. It locates an undamaged bias value from the other values in the same column of the bias map, calls `FEPsciCClkError` to reset the parity plane and T-plane bits, calls `fepAppendRingBuf` to copy a `FEPerrorRec` record to the ring buffer, and returns to the caller the corrected value of the bias map pair.
- `FEPsciTimedError` —resets the appropriate bits in the parity and threshold planes.

1. The distinction made between even and odd pixels is purely for programming efficiency. As a consequence of the FEP hardware architecture, it must access pixel and bias values in pairs.

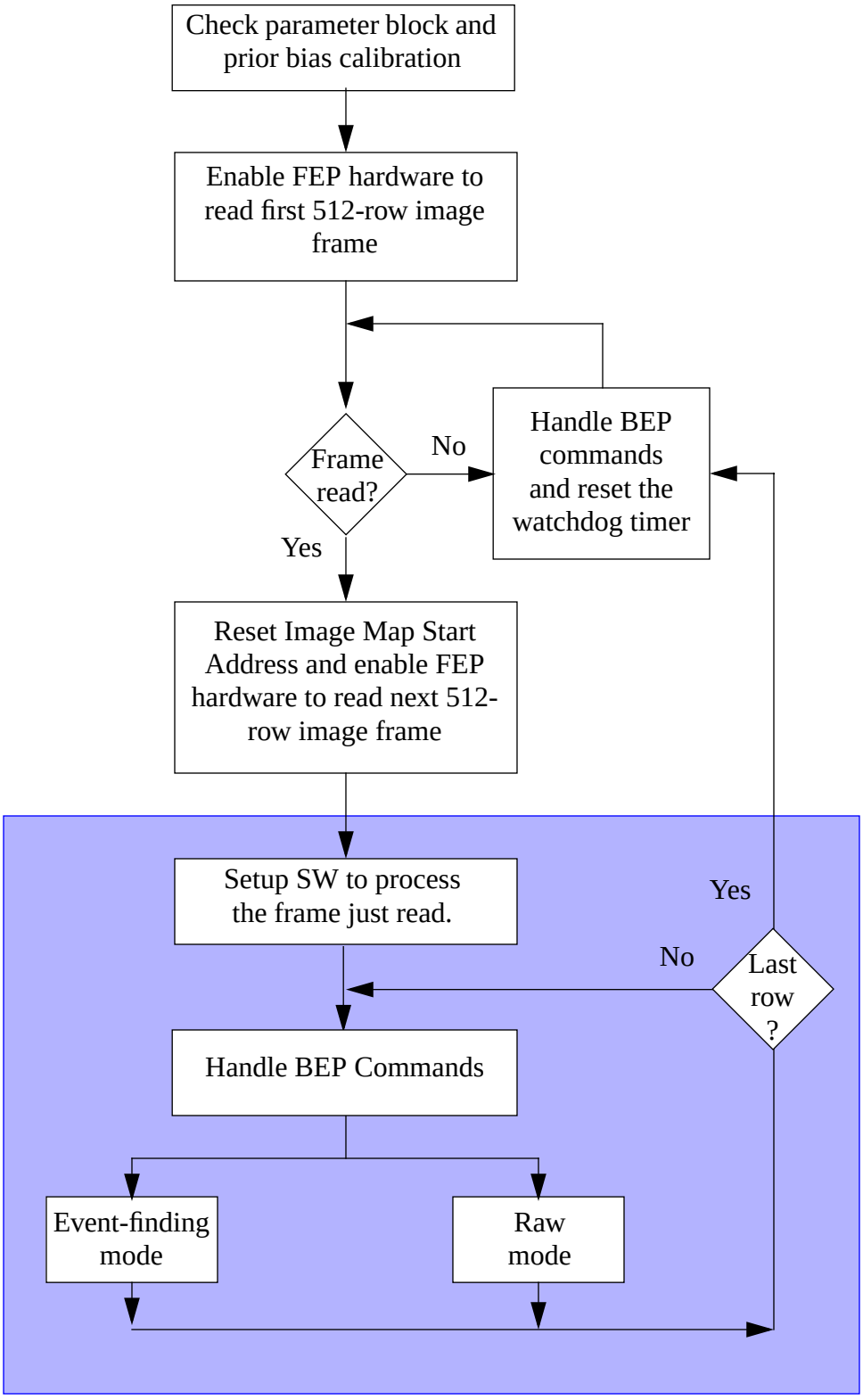


FIGURE 197. The flowchart of fepSciCClk

44.4 Global Variables

The following FEPparm fields, defined in *fepCtl.h* and invariably addressed by the *fp* pointer parameter, are used by all continuously clocked modes:

bepCmd	latest command received from BEP
br	bias calibration parameters
bias0[4]	average nodal overclocks for first bias frame
biassum	sum of the 4 *bias0* values
colshft	bit shift to transform column index to node index
ex	current FEPexpRec record
bias0[4]	copy of *fp->br.bias0* array
dOclk[4]	overclock changes for each output node
expnum	current "exposure" (i.e. VSYNC frame pulse) number
timestamp	microsecond timer value at start of *expnum*
exend	current FEPexpEndRec record
expnum	current "exposure" (i.e. VSYNC frame pulse) number
parityerrs	number of corrected parity errors in the frame
thresholds	number of threshold crossings in the frame
flags	flag bits used:
	FP_SUSPEND — BEP has sent BEP_FEP_CMD_SUSPEND
	FP_TERMINATE — BEP has sent BEP_FEP_CMD_STOP
image	pointer to start of current 512-row frame
lastcol	index of last frame column
nextexpnum	the next frame index that the FEP is to process
quadrants	the number of DEA output nodes being sampled
tp	frame parameter block
initskip	number of initial frames to ignore
ncols	number of CCD columns per output node
noclk	number of overclocks per output node
nrows	number of CCD rows between frame markers
quadcode	output node clocking mode
thresh[4]	threshold values for each output node
type	processing mode, either FEP_CCLK_PARM_1x3 or FEP_CCLK_PARM_RAW.

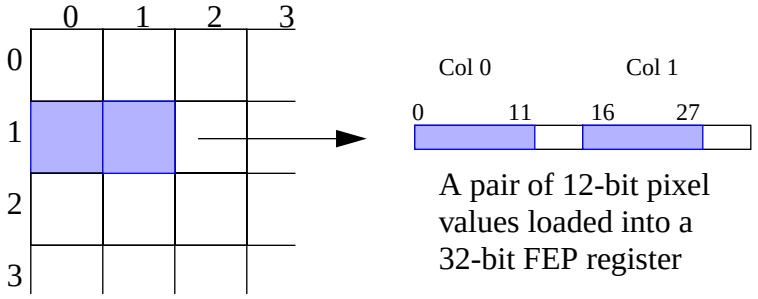
44.5 Scenarios

The following paragraphs describe the basic functions performed in continuously clocked mode. All Continuous Clocking modes wait until an image frame (a block of DEA output pixels delimited by VSYNC flags, typically comprising 512 image rows) is processed and copied to the ring buffer before commanding the hardware thresholder to read the next frame, thereby ensuring that no partial frames are generated.

44.5.1 Use 1: Report 1x3 Events

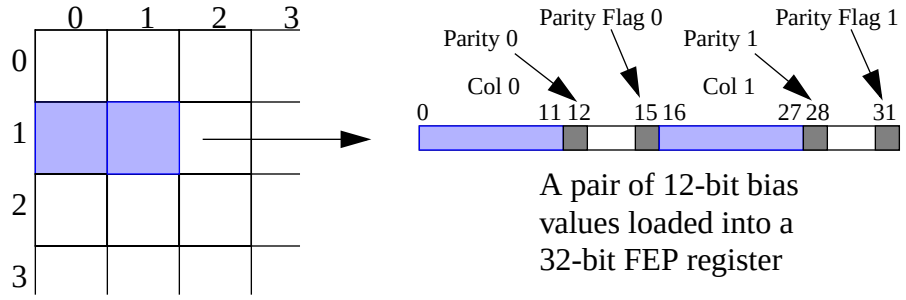
- This mode is characterized by *fp->tp.type* == FEP_CCLK_PARM_1x3. After calling FEPsciCClkInit to check that the bias array contains appropriate values and to load the hardware registers, FEPsciCClk calls fepEnableNextFrame and waits for 512-row frames to arrive. The first *fp->tp.initskip* frames are ignored.
- Once a valid frame arrives, fepSciCClk calls FIOsetImageMapRowStart and fepEnableNextFrame to enable the hardware to write the next 512 rows into the other half of the image map. fepSciCClk then copies the exposure number and arrival time of the previous frame into a FEPexpRec structure and calls fepAppendRingBuf to copy it to the ring buffer. It then calls FEPsciCClkEvent to process the frame.
- FEPsciCClkEvent examines input rows one at a time, calling FIOgetNextCmd to handle any incoming BEP commands. Since the hardware is writing into the *other* 512 rows of image map, it is not necessary to call FIOgetImageMapRowIndex while processing the frame (*cf.* fepSciTimedEvent in Section 42.6.3).
- FEPsciCClkEvent accumulates the current row's overclocks, adding them to the appropriate elements of *oSum*[], indexed by their DEA output nodes. It then inspects the T-plane buffer and calls either FEPsciCClkEvenPixel or FEPsciCClkOddPixel whenever a non-zero bit is found, indicating that the thresholder has located a pixel that exceeds its bias value by *fp->tp.thresh*[nn], where nn is the index of the appropriate DEA output node. Since the 12-bit pixel and bias values are only accessible two at a time via 32-bit CPU instructions, the logic required to inspect even-indexed pixels differs considerably from that needed for odd-indexed pixels, hence the two separate routines.

FIGURE 198. The Relation Between Image Pixels and FEP Register Values



- The two pixel-testing routines examine the pixels on either side of the pixel that triggered the hardware thresholder, as illustrated in Figure 198. Pixels in the bad column list (with a bias value of 4095) are ignored. Otherwise, if the relative value (pixel-minus-bias) of the center pixel exceeds that of the pixels on either side, or equals that of the pixel on its left, `fepAppendRingBuf` is called to report the 3 pixel- and bias-values to the ring buffer in a `FEPeventRec1x3` record.

FIGURE 199. The Relation Between the Bias Map and FEP Register Values



- Whenever either `FEPsciCClkEvenPixel` or `FEPsciCClkOddPixel` load a pair of bias values, as illustrated in Figure 199, they inspect the parity error flags (bits 15 and 31). If either is set, indicating that the parity of the bias value doesn't match the value of the corresponding bit in the bias parity plane, `FEPsciCClkFixBias` is called to (a) locate an undamaged copy of the bias value from the other values in that column of the bias map, (b) reset the bias parity bit and the T-plane bit, and (c) return the corrected bias value. The caller (`FEPsciCClkEvenPixel` or `FEPsciCClkOddPixel`) then stores the corrected value back into the bias map and continues to process the event.
- If all bias values within a given column are found to contain parity errors, the event will be rejected.
- After processing the last row of pixels, `FEPsciCClkEvent` normalizes the *`oSum`*`[]` values and calls `FIOsetThresholdRegister` to update the hardware threshold registers.
- Finally, `fepAppendRingBuf` is called to copy a `FEPexpEndRec` (end-of-frame) record to the ring buffer.

44.5.2 Use 2: Report Raw Pixels

- This mode is characterized by *`fp->tp.type`* == `FEP_CCLK_PARM_RAW`. It begins in an identical manner to the event-finding modes described in the previous sections, except that `fepSciCClk` calls `FEPsciCClkRaw` to process each 512-row frame.
- `FEPsciCClkRaw` copies each row of pixels, and up to 30 overclocks from each DEA output node, to a `FEPeventRecRaw` record and thence to the ring buffer.

44.6 Specification

This section describes the functions that are local to the `fepSciCClk` unit. The only external is `fepSciCClk` itself which is called from `fepCtl`. The `FEPparm` structure is defined in *fepCtl.h*, along with several pixel access macros. The interface to the FEP I/O library is described in Section 39.0, and data and messages exchanged between FEP and BEP are described in Section 4.10.

44.6.1 fepSciCClk()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

`FEPparm *fp`

Description:

`fepSciCClk` is called from `fepCtl` with a single argument: the pointer `fp` to the `fepParm` structure. It is responsible for all FEP actions associated with a timed exposure science run, except for bias calibration. In particular, it controls the double-buffering of input data—the image map is split logically into two 512-row buffers and the FEP hardware is told to write into one while the software is processing the other. `fepSciCClk` performs the following actions:

i) Calls `FEPsciCClkInit` to validate the BEP's parameter block, *`fp->tp`*, for the current mode, to verify that pixel biases have been computed, and to initialize mode-dependent variables. If `FEPsciCClkInit` returns `FEP_CMD_NOERR`, `fepSciTimed` calls `fepAckCmd` and continues. Otherwise, it calls `fepNackCmd` to pass the `fepCmdRetCode` error code to the BEP, indicating that the command has failed, and `fepSciCClk` then returns to `fepCtl`. Error codes are defined in *fepBep.h* (see Section 4.10).

ii) Calls `fepEnableNextFrame` to enable the hardware to read the first 512 rows of image pixels into the image buffer.

iii) Loops over frames of 512 rows of image pixels.

iv) Enters a loop (see the upper half of Figure 197), calling `FIOgetExpInfo` until the exposure numbered `fp->nextexpnum` is encountered (or exceeded). During this loop, calls are made to `FIOgetNextCmd` to handle any incoming BEP commands, and to `FIOtouchWatchdog` to prevent the Watchdog Timer from expiring while waiting for that particular frame. *`fp->flags`* is inspected: if the `FP_TERMINATE` bit is set, `fepSciCClk`

returns immediately—the science run is over; if the FP_SUSPEND bit is set, fepSciCClk loops until the BEP sends it a BEP_FEP_CMD_RESUME command.

v) Calls FIOsetImageMapRowStart and fepEnableNextFrame to enable the hardware to read the next 512 rows into the *other* half of the image buffer.

vi) Calls either FEPsciCClkEvent or FEPsciCClkRaw (according to the value of *fp->tp.type*) to process the 512 rows that were read by the hardware *prior* to step v).

vii) Branches back to step iii), above. The loop over input frames continues until FIOgetNextCmd receives a BEP_FEP_CMD_STOP command from the BEP, which sets the FP_TERMINATE bit in *fp->flags*. fepSciCClk will finish processing the current frame before returning to fepCtl.

44.6.2 FEPsciCClkInit()

#include fepCtl.h

Scope: Science

Return type: fepCmdRetCode

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciCClk` to verify the contents of the `FEPparmBlock`, *`fp->tp`*, and to check whether a bias calibration has been performed. It also performs any necessary mode-dependent initialization. When in event-detection mode (*`fp->tp.type`* == `FEP_CCLK_PARM_1x3`), hardware thresholding and bias parity error detection are enabled. Otherwise, both are disabled. Overclock processing is always enabled.

The following *`fp->tp`* and *`fp->br`* fields are checked:

Tests Applied	*Error code returned if the test fails*
quadcode ∈ `fepQuadCode`	`FEP_CMD_ERR_QUAD_CODE`
$0 <$ *nrows* $\leq$ `CCLK_NROWS`	`FEP_CMD_ERR_NROWS`
$0 <$ *ncols*, and *ncols* even	`FEP_CMD_ERR_NCOLS`
quadrants * *ncols* $\leq$ `MAX_NCOLS`	`FEP_CMD_ERR_NCOLS`
noclk $\leq$ `MAX_NOCLK`	`FEP_CMD_ERR_NOCLK`
$\sum_{i=0}^{3} br.bias0[i] \equiv br.biassum$	`FEP_CMD_ERR_NO_BIAS`

If an error is detected, `FEPsciCClkInit` returns the appropriate *`fepCmdRetCode`* value, as defined in *fepBep.h* (see Section 4.10); otherwise it returns `FEP_CMD_NOERR`.

44.6.3 FEPsciCClkEvent()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciCClk` to process a single continuously clocked frame. It initializes the following row pointers to the addresses of data structures used by the hardware thresholder:

pTPlane	points to first data bit in T-plane buffer
pOclk	points to start of overclock buffer
fp->image	points to start of first data row in image buffer

and these pointers will be advanced from row to row. Note that, in step with the double buffering, *pTPlane* and *fp->image* will alternately point to the start and to the middle of their respective buffers. The four `oSum[]` overclock accumulators (one per DEA output node) are zeroed.

`FEPsciCClkEvent` then loops over CCD pixel rows. Once per row, it calls `FIOgetNextCmd` to see whether the BEP is trying to command the FEP. A returned value of `TRUE` signals that a science command has been received (utility commands will be executed within `FIOgetNextCmd` and will return `FALSE`), and a call will be made to `fepHandleCmd` to process it. NOTE: only one call is made to `FIOgetNextCmd` per input row.

The function then processes the row data. It adds the overclocks to the `oSum` accumulators, and then examines the T-plane, 32 bits at a time, until a non-zero value is found—indicating that the hardware detected a threshold crossing or bias map parity error. Because the 12-bit pixel and bias values must be loaded in pairs, the 32 T-plane mask bits are tested two at a time—if an even-offset bit has been set, `FEPsciCClkEvenPixel` is called; otherwise `FEPsciCClkOddPixel` is called.

After processing the image row, `FEPsciCClkEvent` increments the row pointers. After the last row of the frame, the *oSum* overclock accumulators are normalized and used to derive thresholds for the next exposure, which are communicated to the hardware by calls to `FIOsetThresholdRegister` for each DEA output node.

44.6.4 FEPsciCClkEvenPixel()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from FEPsciCClkEvent for every bit set in the T-plane whose bit offset is even. The associated pixel is located in row *irow*, column *icol*, and *fp->image* points to the first pair of pixels in that row. The corresponding pair of bias values are to be found in *fp->image*[BIAS_OFFSET].

FEPsciCClkEvenPixel first loads the 32-bit pixel and bias fields that contain the central pixel. Since the FEP is a little-endian processor, and the central pixel is stored in the lower address pair of bytes, pixel and bias will be loaded into the least-significant 16 bits of each 32-bit variable. (The most significant 16 bits of these variables will contain the pixel and bias for row *irow* and column *icol*+1.)

The pair of bias values is now validated—the bias parity flags (bits 15 and 31) are tested to determine whether a parity error has occurred in either of those locations in the bias map. If so, a call is made to FEPsciCClkFixBias to handle the problem, and the corrected 32-bit bias pair is written back to the bias map. Unlike the situation in timed exposure mode, when the software is merely able to report the damaged pixel, in continuous clocking mode the value can actually be repaired since each row of the bias map should be identical. Therefore, once the bias value has been updated, FEPsciCClkEvenPixel continues execution with the corrected value.

Next, a test is made to determine whether this central pixel lies on the left- or right-hand edge of the CCD, or if its bias value is PIXEL_BAD (indicating that the pixel is a member of the bad-pixel list). In any of these situations, the pixel is ignored.

The 12-bit value of the central pixel is saved in ev.p[1] of a local FEPeventRec1x3 structure, and the corresponding 12-bit bias value in *ev.b*[1]. To assist in later FEPsciCClkPixTest calls, the variable *val* is set equal to the difference between the center pixel value and its bias values, as dis-

cussed in Section 44.6.6.

The center pixel is now compared to those lying to its left and right, using `FEPsciCClkPixTest`, which also saves the pixel and bias values in the appropriate elements of the *`ev.p`* and *`ev.b`* arrays. If a test fails, i.e. if the central pixel isn't a local maximum, the function returns. When testing the pixel lying to the left, *`val`* must be adjusted for any change in average overclock if the two pixels were generated by different CCD output nodes. This change is simply the difference between the corresponding node values in the *`fp->ex.dOclk`*`[]` array.

If the pixels on either side of the central pixel survive the `FEPsciCClkPixTest` criteria, the 1x3 pixel and bias arrays in the *`ev`* structure will have been loaded. `FEPtestEvenPixel` then calls `fepAppendRingBuf` to write the `FEPeventRec1x3` record to the FEP-BEP ring buffer.

44.6.5 FEPsciCClkOddPixel()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from FEPsciCClkEvent for every bit set in the T-plane whose bit offset is odd. The associated pixel is located in row *irow*, column *icol*, and *fp->image* points to the first pair of pixels in that row. The corresponding pair of bias values are to be found in *fp->image*[BIAS_OFFSET].

FEPsciCClkOddPixel first loads the 32-bit pixel and bias fields that contain the central pixel. Since the FEP is a little-endian processor, and the central pixel is stored in the upper address pair of bytes, pixel and bias will be loaded into the most-significant 16 bits of each 32-bit variable. (The least significant 16 bits of these variables will contain the pixel and bias for row *irow* and column *icol*-1.)

The pair of bias values is now validated—the bias parity flags (bits 15 and 31) are tested to determine whether a parity error has occurred in either of those locations in the bias map. If so, a call is made to FEPsciCClkFixBias to handle the problem, the corrected 32-bit bias pair is written back to the bias map. Unlike the situation in timed exposure mode, when the software is merely able to report the damaged pixel, in continuous clocking mode the value can actually be repaired since each row of the bias map should be identical. Therefore, once the bias value has been updated, FEPsciCClkOddPixel continues execution with the corrected value.

Next, a test is made to determine whether this central pixel lies on the left- or right-hand edge of the CCD, or if its bias value is PIXEL_BAD (indicating that the pixel is a member of the bad-pixel list). In any of these situations, the pixel is ignored.

The 12-bit value of the central pixel is saved in ev.p[1] of a local FEPeventRec1x3 structure, and the corresponding 12-bit bias value in *ev.b*[1]. To assist in later FEPsciCClkPixTest calls, the variable *val* is set equal to the difference between the center pixel value and its bias value, as dis-

cussed in Section 44.6.6.

The center pixel is now compared to those lying to its left and right, using `FEPsciCClkPixTest`, which also saves the pixel and bias values in the appropriate elements of the *`ev.p`* and *`ev.b`* arrays. If a test fails, i.e. if the central pixel isn't a local maximum, the function returns. When testing the pixel lying to the right, *`val`* must be adjusted for any change in average overclock if the two pixels were generated by different CCD output nodes. This change is simply the difference between the corresponding node values in the *`fp->ex.dOclk`*`[]` array.

If the pixels on either side of the central pixel survive the `FEPsciCClkPixTest` criteria, the 1x3 pixel and bias arrays in the *`ev`* structure will have been loaded. `FEPsciCClkOddPixel` then calls `fepAppendRingBuf` to write the `FEPeventRec1x3` record to the FEP-BEP ring buffer.

44.6.6 FEPsciCClkPixTest

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

*FEPeventRec1x3 *ev*

unsigned dcol

int val

unsigned pixel

unsigned mode

unsigned bias

Description:

This is an `inline` function that is used several times within the `FEPsciCClkEvenPixel` and `FEPsciCClkOddPixel` functions. It masks the 12 low-order bits of `pixel` and `bias` and stores them in *p*[*dcol*] and b[*dcol*] in the `ev` structure. It then compares the value of (*pixel - bias*) against `val`.

`FEPsciCClkPixTest` evaluates to `TRUE` when the value of the pixel at *dcol* is inconsistent with a legal event, or `FALSE` when it isn't. Illegal events are those for which the corresponding bias value is `PIXEL_BAD`, i.e . when the corresponding pixel is a member of the bad column list, or those that pass the test defined by the *mode* parameter, which is either `LE` ("less-than-or-equal"), in which an illegal pixel satisfies

val <= (pixel - bias)

or LT ("less than"), when an illegal pixel satisfies

val < (pixel - bias)

dcol must be in the range `0-2`; the "center" pixel has *dcol*=1; *dcol*=0 refers to the column before the center pixel, *dcol*=2 to the column after. Constants `PIXEL_MASK`, and `PIXEL_BAD` are defined within *fepCtl.h*, and *val* and *ev* are local variables.

44.6.7 FEPsciCClkFixBias()

#include fepCtl.h

Scope: Science

Return type: unsigned

Arguments

unsigned bias

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from `FEPsciCClkEvenPixel` or `FEPsciCClkOddPixel`, when a parity error is detected in either or both of the halves of a 32-bit `bias` word. Each corrupted 12-bit value is replaced by the first non-corrupted value in the same column *`icol`* of the bias map, and `FEPsciTimedError` is called (twice if both 12-bit values are bad) to reset the appropriate bits in the Bias Parity Plane and T-plane. It then calls `fepAppendRingBuf` to send a message of type `FEP_ERROR_REC` to the BEP containing *`irow`*, *`icol`*, the uncorrected *`bias`* value, and the current exposure number in *`fp->ex.expnum`*. Finally, `FEPsciCClkFixBias` returns the fixed-up 32-bit `bias` word to the caller, which has the responsibility for saving it back in the appropriate location in the bias map. NOTE: since `bias` represents a pair of values, *`icol`* must necessarily be even.

44.6.8 FEPsciCClkError()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

unsigned val

unsigned irow

unsigned icol

*FEPparm *fp*

Description:

This function is called from `FEPsciCClkFixBias` when a parity error is detected in row *irow*, column *icol* of the bias map. It computes the parity of the 12-bit bias value, *val*, and resets the relevant bit in the Bias Parity Plane, turns off the corresponding bit in the T-plane, and increments the bias error counter, *fp->exend.parityerrs*.

44.6.9 FEPsciCClkRaw()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from `fepSciCClk` to process a single raw CCD frame. It initializes the following pointers to the addresses of data structures used by the hardware thresholder:

pOclk	points to start of overclock buffer
fp->image	points to start of first data row in image buffer

and these pointers will be advanced from row to row. Note that, in step with the double buffering, *fp->image* will alternately point to the start and to the middle of the image buffer.

`FEPsciCClkRaw` then loops over CCD pixel rows. Once per row, it calls `FIOgetNextCmd` to see whether the BEP is trying to command the FEP. A returned value of `TRUE` signals that a science command has been received (utility commands will be executed within `FIOgetNextCmd` and will return `FALSE`), and a call will be made to `fepHandleCmd` to process it. NOTE: only one call is made to `FIOgetNextCmd` per incoming pixel row.

The function then processes the row data, copying the raw pixels and overclocks to a `FEPeventRecRaw` structure and then calling `fepAppendRingBuf` to copy it to the ring buffer.

45.0 FEP Continuously Clocked Bias Calibration (36-53227 A)

45.1 Purpose

The ***fepCClkBias*** module, executing in the FEP, calibrates the bias map for subsequent continuously-clocked science processing. It is called from ***fepCtl*** with a single argument, `fp`, a pointer to the `fepParm` structure. Before invoking ***fepCClkBias***, the FEP must be commanded to load a continuously-clocked `FEPparmBlock` into *`fp->tp`*.

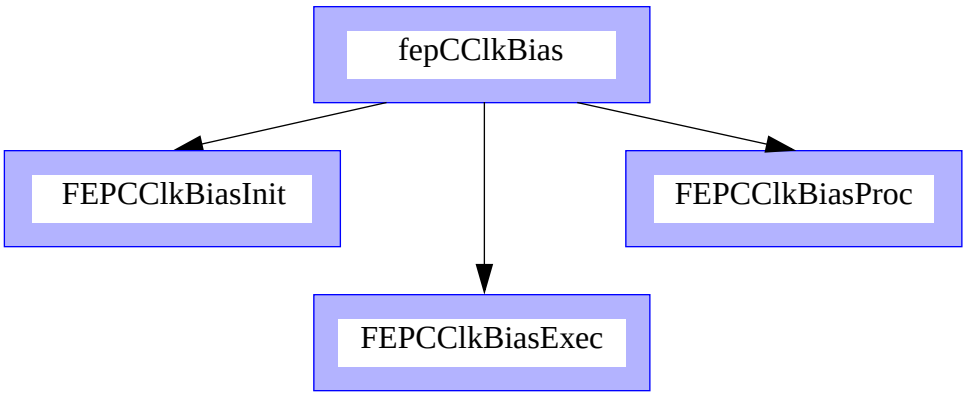
In continuous clocking mode, all pixels within the same column have moved through all rows of CCD image and frame stores, and are therefore characterized by identical values of bias threshold. However, the FEP hardware thresholder requires the bias map to be 2–dimensional as in timed exposure more, although all entries in a given column will be identical. It is therefore particularly simple to compute the continuously clocked bias map —the image map is filled with *N* consecutive rows of pixels (typically 1024), and the bias threshold for each column is determined from the *N* values. The values are replicated in each row of the bias map. If a bias value is subsequently upset by a high-energy event and is detected by its incorrect parity, the correct value can be retrieved from the remaining values in the same column of the bias map.

45.2 Uses

The ***fepCClkBias*** function has a single mode of operation:

Use 1:: Calculate a continuously-clocked bias map.

FIGURE 200. fepCClkBias Structure



45.3 Organization

All interactions with the BEP and with FEP hardware are made through ***fepio*** library functions described in Section 39.0. The commands that are passed between FEP and BEP are defined in that document and in Section 4.10. ***fepCClkBias*** makes use of the following functions which call each other in the manner shown in Figure 1:

- ***FEPCClkBiasInit***—is called from ***fepCClkBias*** to validate the FEPparmBlock, *fp->tp*, and perform all necessary initialization.
- ***FEPCClkBiasExec***—is called from ***fepCClkBias*** to process a pair of 512-row image frames. It sums the overclocks and leaves the image pixels in the image buffer.
- ***FEPCClkBiasProc***—is called from ***fepCClkBias*** to process each column of the image buffer. It copies columns of pixels into a vector, calls ***mean*** or ***fractile*** (defined in the ***fepTimedBias*** module) to compute the bias value, and stores the result into each element of the corresponding column of the bias map. At the same time, it construct a bias parity table in which each bit represents the parity (EVEN=0, ODD=1) of the corresponding bias map value.

45.4 Global Variables

The following FEPparm fields, defined in *fepCtl.h* and *fepBep.h*, are invariably addressed by the *fp* pointer parameter, are used by the bias calculation:

bepCmd	latest command received from BEP	
br	pointer to bias calibration parameters	
bias0[4]	average overclocks for first bias frame	
biassum	sum of the 4 *bias0* values	
ex	current FEPexpRec record	
expnum	current “exposure” number	
timestamp	microsecond timer value at start of *expnum*	
expcount	number of 512-row bias frames processed	
fepStatus	FEP status reported to BEP	
biasflag	=1 if bias has been computed	
flags	flag bits:	
	FP_SUSPEND	BEP_FEP_CMD_SUSPEND sent
	FP_PAST_EOR	FEP hardware has finished with the current frame
	FP_TERMINATE	BEP has sent BEP_FEP_CMD_STOP
	FP_DONE	BEP is terminating normally
nextexpnum	the next exposure index that the FEP is to process	
quadrants	the number of DEA output nodes being sampled	
tp	exposure parameter block (see Table 48)	
bparm[5]	mode-dependent parameters	
initskip	number of initial frames to ignore	
ncols	number of CCD columns per output node	
noclk	number of overclocks per output node	
nrows	number of CCD rows between frame markers	
quadcode	output node clocking mode	

45.5 Scenario

The following paragraphs describe the basic functions performed by ***fepCClkBias*** during continuously clocked bias calibrations, which are determined by the fields in the parameter block, `fp->tp` shown in Table 48.

TABLE 48. Parameters used by fepCClkBias

Field Type	*Field Name*	*Description*
unsigned	*nrows*	Number of pixel rows between frame markers.
unsigned	*ncols*	Number of pixels per output node per row
fepQuadCode	*quadcode*	Output node configuration, i.e., ABCD, AC, or BD.
unsigned	*noclk*	Number of overclocks per row per output node
int	*bparm*[1]	=0 to use the Interated Mean algorithm, ***mean*** =1 to use the Fractile algorithm, ***fractile***
int	*bparm*[2]	For ***mean***, specifies σ rejection criterion. For ***fractile***, index of sorted pixel array.

The bias calibration algorithms themselves are presented in some detail in the ACIS report entitled "*CCD Bias Level Determination*" by Rita Somigliana and Peter Ford, ACIS part #36–56101–02, MIT CSR, Revision 2.1, June 19, 1995.

45.6 Algorithms

45.6.1 The Iterated Mean Algorithm

This algorithm, described in Section 3.2.4 of 36–56101–02, is implemented by the function ***mean*** (see page 1265 of the current document). It takes the N pixel values p_i, i=0,N–1, (N is `2*fp->tp.nrows`, usually 1024 in continuously clocked mode), and computes their mean value $\bar{p}$ and variance σ^2:

$$\bar{p} = \frac{1}{N}\sum_{i=0}^{N-1} p_i \qquad \text{(EQ 9)}$$

If the value of *BPARM*[2] is zero, ***mean*** returns $\bar{p}$ as the bias value. Otherwise, it inspects the p_i and removes any that do not satisfy the condition

$$|p_i| \le (\sigma \cdot BPARM[2]) \qquad \text{(EQ 10)}$$

$$\sigma^2 = \frac{1}{N-1}\sum_{i=0}^{N-1} (p_i - \bar{p})^2 \qquad \text{(EQ 11)}$$

Finally, it recomputes $\bar{p}$ from the remaining p_i using Eq. 9, and returns it as the bias value.

45.6.2 The Fractile Algorithm

This algorithm, which is a generalization of the Median method described in Section 3.2.5 of 36–56101–02, is implemented in the function ***fractile*** (see page 1266 of the current document). It takes the N pixel values p_i, i=0,N–1, sorts them into ascending order, and returns the value indexed by the value of `bparm[2]`. For instance, if N were 11, and `bparm[2]` were 5, and the pixel values p_i were,

```
212 216 205 1041 208 217 211 214 215 206 210
```

their bias value would be 212, since, when the values are sorted into ascending order,

```
205 206 208 210 211 212 214 215 216 217 1041
```

that is the value of the element p_5. In practice, N is `2*fp->tp.nrows`, usually 1024.

45.6.3 Use 1: Calculate a Continuously-Clocked Bias Map

fepCClkBias calls ***FEPCClkBiasInit*** to validate the `fp->tp` parameter block, to initialize the parity table (`fp->parity`), and set the hardware registers to write the first half-frame (512 rows) of image pixels into the first half of the image buffer. It then loops over calls to ***FIOgetExpInfo***, waiting for the image frames to be received from the DEA. The first `fp->tp.initskip` frames are ignored.

Immediately a valid frame arrives, ***fepCClkBias*** resets the hardware registers to cause the second set of 512 rows to be written into the remainder of the image buffer. It then calls ***FEPCClkBiasExec*** to monitor the arrival of the `2*fp->tp.nrows` rows and accumulating overclock values. Once the last row has been processed, ***FEPCClkBiasExec*** saves the average overclock for each CCD quadrant and returns. While waiting for each row to arrive, ***FEPCClkBiasExec*** calls ***FIOgetNextCmd*** to see whether a command has arrived from the BEP. If it has, ***FEPCClkBiasExec*** calls ***fepHandleCmd*** and then tests the `FP_TERMINATE` bit in `fp->flags`. If the latter has been set, as a result of receiving a `BEP_FEP_CMD_STOP` command, ***FEPCClkBiasExec*** terminates immediately and ***fepCClkBias*** marks the bias map as bad.

fepCClkBias then calls ***FEPCClkBiasProc*** to calculate the bias for each CCD column, using either the “iterated mean” algorithm, or the “fractile” algorithm. These are discussed in detail in Section 3.2 of the *CCD Bias Level Determination* report described in Section 45.5 above. Once computed, the bias values are written into all elements of that column of the bias map. Their parity is also calculated and written into the bias parity buffer.

45.7 Specification

This section describes the functions that are local to the ***fepCClkBias*** module. The only external is ***fepCClkBias*** itself which is called from ***fepCtl***. The `FEPparm` structure is defined in *fepCtl.h* and *fepBep.h*, along with several pixel access macros. The interface to the FEP I/O library is described in Section 39.0, and data and messages exchanged between FEP and BEP are described in Section 4.10.

45.7.1 fepCClkBias()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

```
FEPparm *fp
```

Description:

fepCClkBias is called from ***fepCtl*** with a single argument: the pointer *fp* to the `fepParm` structure. It is responsible for all FEP actions associated with a timed exposure bias calibration. It performs the following actions:

- Calls ***FEPCClkBiasInit*** to initialize various `fepParm` fields and hardware registers. It tells the hardware to begin writing the first frame at the top of the image map. If ***FEPCClkBiasInit*** returns `FEP_CMD_NOERR`, ***fepCClkBias*** calls ***fepAckCmd*** and continues. Otherwise, it calls ***fepNackCmd*** to pass the error code to the BEP, indicating that the command has failed, and ***fepCClkBias*** then returns to ***fepCtl***.
- Waits for the first image frame to arrive from the DEA. While waiting, ***fepCClkBias*** executes a tight loop, alternatively calling ***FIOtouchWatchdog*** to keep the watchdog timer alive, and ***FIOgetNextCmd*** to intercept and process commands from the BEP. Once ***FIOgetExpInfo*** indicates that the frame is being copied into the image map, ***fepCClkBias*** calls ***FIOsetImageMapRowStart*** to tell the FEP hardware to write the second frame into the lower half of the image map, starting at row *fp->tp.nrows*, i.e. directly after the first frame.
- Calls ***FEPCClkBiasExec*** to monitor the incoming image rows and sum the overclocks.
- Calls ***FEPCClkBiasProc*** to compute the bias values, and to set the bias parity buffer.

- Unless the FP_TERMINATE bit is set in *fp->flags*, indicating that the BEP has issued a BEP_FEP_CMD_STOP command, it marks the bias map as "good" by setting *fp->biasmode* to TRUE and *fp->br.biassum* to the sum of the 4 elements in the *fp->br.bias0* array, and it saves these values in I-cache via a call to ***FIOsetBiasConfig***.

45.7.2 FEPCClkBiasExec()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***fepCCLKBias*** to process a single frame. It loops over incoming rows of image pixels, calling ***FIOgetNextCmd*** once per row to catch incoming commands from the BEP and, when detected, ***fepHandleCmd*** is called to process them. ***FEPCClkBiasExec*** then sums the overclocks. The process is terminated prematurely if a BEP_FEP_CMD_STOP command is received from the BEP, in which case the FP_TERMINATE flag is set in *fp->flags*.

Once a pair of frames (2**fp->tp.nrows* *rows*) has been processed, the overclock values are summed and saved in *fp->br.bias0*.

45.7.3 FEPCClkBiasInit()

#include fepCtl.h

Scope: Science

Return type: fepCmdRetCode

Arguments

*FEPparm *fp*

Description:

This function is called from ***fepCClkBias*** to verify the contents of the FEPparmBlock, *fp->tp*, and to perform the following initializations:

- *fp->quadrants* is set to to 2 or 4, depending on the value of *fp->tp.quadcode*.
- FEP hardware registers are set by calls to ***FIOsetCcdRowStart***, ***FIOsetImageMapRowStart***, ***FIOsetImageMapRowLength***, and ***fepSetAddrMode***. Thresholding and bias parity error detection are disabled. Overclock processing is enabled.

The bias parameter structure, *fp->br* is initialized with “bad” values, i.e. *fp->br.biassum* is set to the unphysical value ~0, and ***FIOsetBiasConfig*** is called to save this in I-cache.

If an error is detected, ***FEPCClkBiasInit*** returns a *fepCmdRetCode* code as defined in *fepBep.h*; otherwise it returns FEP_CMD_NOERR.

45.7.4 FEPCClkBiasProc()

#include fepCtl.h

Scope: Science

Return type: void

Arguments

*FEPparm *fp*

Description:

This function is called from ***fepCClkBias*** once the image map has been filled with two frames of pixel rows. It begins by computing a table of 4096 elements, each either PARITY_EVEN or PARITY_ODD, according to the parity of the integers 0 through 4095. For instance, the number fifteen is represented by the bit pattern 01111, which contains an even number of '1's. Its parity is therefore even, so *fp->parity*[15] = PARITY_EVEN.

The image pixels are then examined column by column, a bias value is computed for each column and stored in the corresponding column of the bias map. The bias values are calculated by copying a pair of pixel columns to a pair of value arrays, *Data0* and *Data1*, located in D-cache (for access speed). The algorithm used to compute the bias values is determined by the value of *fp->tp.bparm*[1].

bparm[1]=0: ***mean*** is called to compute the mean of the pixel values,as described in Section 45.6. When *fp->tp.bparm*[2] is zero, this becomes the bias map value. When *fp->tp.bparm*[2] is non-zero, pixels with values that differ from the zeroth-order mean by more than *fp->tp.bparm*[2] times the RMS variance of the values are eliminated, and the mean of the remainder becomes the bias map value.

bparm[1]=1: ***fractile*** is called to sort the pixel values into ascending order. The element in this sorted list indexed by *fp->tp.bparm*[2] becomes the bias map value, as described in Section 45.6.2.

Once a pair of bias values has been computed, they are stored in the corresponding columns of the bias map. Then their parities are computed and, every 32 columns, a column of 32-bit parity flags is stored in the bias parity map. It is important to store the bias values and their parity flags as soon as they are calculated since their maps are located in bulk memory which is subject to single-event upsets (SEUs). By saving the values in all rows simultaneously, elements that have suffered SEUs can be replaced at a later time by undamaged duplicate values.

Appendix A - FEP Timing Budget

Purpose

This section describes the timing budget for the science and bias processing modes of the Front End Processor. It details the conditions under which the data throughput requirements of the ACIS Contract End Item Specification (§3.1.3.2a of 36-01101-06) are to be met. These are stated as follows:

- "The digital processing system shall be capable of continuous operation in event mode with any input data stream with all of the following characteristics: 1) containing at least 1000 pixels above threshold per second per CCD data stream (due to valid X-ray events, background events, and hot pixels); and ii) containing a total valid X-ray event rate (in all six CCD data streams) of a rate of at least 750 events per second."

These requirements only apply to *event* mode, not to either *histogram* or *raw* mode, since these are for diagnostic purposes only. However, in the interest of completeness, timing budgets have been constructed for all modes.

Overall Approach

Since the FEP software has been designed according to function rather than performance, timing requirements were not imposed *a priori* in the design, but rather as a set of *a posteriori* qualifications to which the prototype designs were subjected. The process is went follows:

- A detailed design of a particular FEP science mode was performed and a preliminary design document written.
- Prototype C-language source code was written.
- The design document and prototype code were reviewed. Obvious errors were corrected and re-design suggestions implemented.
- The prototype code was converted to assembler instructions by the *gcc* compiler (-S option) on an R3000 DecStation using maximal optimization.
- The assembler code was inspected, and the number of machine instructions required to perform the various stages of the FEP science modes were counted, along with the number of accesses (always entire 32-bit words) to non-cache memory.

Results

The results are shown in Table 49. The "cycles" column lists the number of machine instructions that were generated for each function, and the "access" column lists the number of 32-bit accesses to non-cache memory, assuming each row contains 1024 pixels and 128 overclocks (a maximum of 32 from each of 4 output nodes).

The rightmost column estimates the execution time of each function, assuming that all cache accesses occupy a single machine cycle (0.1 μsec), and that non-cache accesses require an additional 2 cycles to perform.

TABLE 49. FEP machine cycles and non-cache memory accesses

Science Mode	*Function*	*Sub-function*	*Cycles*	*Access*	*Msec*
Timed Event	Set Up		237		0.023
	Per CCD row	Set Up	71		0.007
		Overclocks	1600	64	0.173
		Threshold detection	7424	32	0.755
		Total	9095	96	0.935
	Per threshold crossing	Set Up	57	2	0.006
		Detect event	179	12	0.021
		Copy to BEP	152	11	0.018
		Total	388	25	0.045
	Per CCD frame[a]	Set Up	237		0.023
		Total (msec per frame)			**1074.500**
Raw Pixels	Set Up		150		0.015
	Per CCD row	Set Up	57		0.006
		Format pixels	3584	512	0.461
		Format overclocks	768	64	0.090
		Copy to BEP	5762	574	0.691
		Total	10114	1150	1.248
	Per CCD frame[b]	Set Up	119		0.012
		Total (msec per frame)			**1277.970**
Histogram	Set Up		162		0.017
	Per CCD row	Set Up	45		0.005
		Pixel histogram	7860	512	0.885
		Average overclocks	5376	64	0.551
		Total	13281	576	1.441
	Report buffer	Clear buffer	65561		6.556
		Copy to BEP	163992	16395	19.679
	Per CCD frame[c]	Set Up	124		0.013
		Total (msec per frame)			**1475.600**

a. assuming 1024 rows per frame and 2600 threshold crossings, each a local maximum

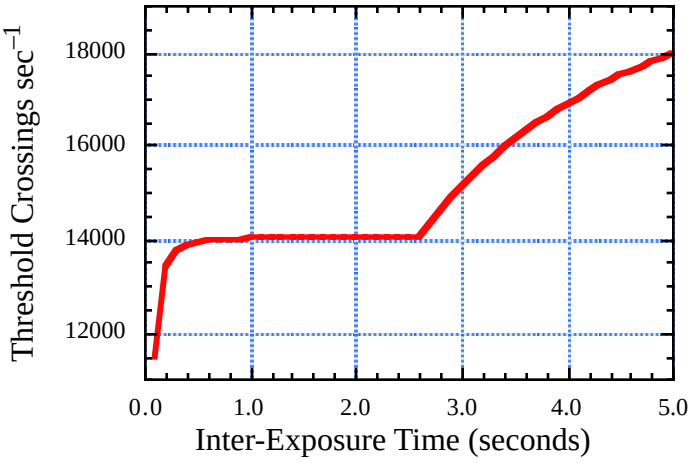
b. assuming 1024 rows per CCD frame

c. assuming 1024 rows per CCD frame, with the histogram arrays in cache memory

The times quoted in Table 49 were based on the most pessimistic assumptions for a full-frame **timed exposure**—a readout time of 2.6 seconds, and hence a total of 2600 threshold crossings, each of which is a local maximum and therefore causes an event record to be sent to the BEP. The figure of 1.074 seconds of CPU time per frame represents a perfor-

mance margin of 59%. Note, however, that most of the processor time is spent in locating the threshold crossings—the overhead incurred in determining whether a particular crossing is a local minimum, and then copying the event record to the ring buffer, is very small. Inverting the argument, Figure 201 shows how the maximum number of threshold crossings per second is expected to vary with the time between exposures. The break at 2.6 seconds marks the point at which shorter readout times can only be achieved by clocking the CCD in sub-frame readout mode.

FIGURE 201. Threshold Crossings vs. Inter-Exposure Time



There are two conditions under which the CEI specification may not be met. First, when partial frames are read from the CCDs (sub-frame mode) and the inter-exposure time is less than about 0.2 second. In this situation, the need to synchronize the FEP processors via frequent BEP handshakes, coupled with the overhead of generating exposure records, will cause the performance to deteriorate in a TBD manner.

Second, when a disproportionate number of threshold crossings are concentrated at the bottom of the image frame, i.e. the part that is examined last, the FEP may not have sufficient time to process the pixels before they are overwritten by the next frame. This would cause the BEP to command all FEPs to skip a frame. If this condition persists, the overall duty cycle would be halved.

Table 49 also shows that **histogram mode** possesses a 43% margin in its ability to process full 1024-line image frames at an inter-exposure time of 2.6 seconds. However, each set of histogram arrays occupies 66 Kbytes, which will eventually saturate the 24 Kbaud telemetry downlink if reported more frequently than once per 22 seconds (once per 18 minutes in 500 baud “next-in-line” mode). Similarly, the FEPs possess adequate margin in **raw mode**, but will quickly fill available buffer space in the BEP and FEPs.

Appendix B - TBD List