XMM OM

X-ray Multi-mirror Mission Optical Monitor

User Manual Part 1B - Experiment On-Board Software - Digital Processing Unit

XMM-OM/UCSB/ML/0012.1

Jeonghee Rho
University of California, Santa Barbara

Scott D. Horner
Pennsylvania State University

James R. M. Klarkowski
Sandia National Laboratories

Cheng Ho
Los Alamos National Laboratory

Alan D. Welty
Pennsylvania State University

Tim Sasseen
University of California, Santa Barbara

22 June 98

(printed 29 June 1998)

University of California, Santa Barbara

Department of Physics

Contents

B Abstract

The Digital Processing Unit (DPU) is responsible for the on-spacecraft data processing of the detector data on the X-ray Multi-mirror Mission Optical Monitor (^{XMM}OM). This document is a description of the framework for the flight on-spacecraft software. Section 1 is an introduction to this document, §2 is a description of the DPU and the tasks it is required to perform, and §3 describes the DPU's interfaces to related systems. A description of the system design methods, and the software architecture data flow diagrams and state transition diagrams is in §4. Functional descriptions of the each software component are separated into three sub-sections in §5: White DSP, Red DSP, and Blue DSP softwares. The library header files of the DPU software are described in §6. Appendix A lists the acronyms and abbreviations used in this document, Appendix B is a lexicon of terms used in describing the DPU operation, and Appendix C contains a description of the compiling utilities and procedures used to compile the flight software and prepare it for loading into the DPU memory.

C Document Change Record

Date	Version	Comments
15 August 94	0003.draft0.1	Initial Draft
15 November 94	0003.draft0.2	US Internal Draft
15 March 95	0003.draft0.3	OM Internal Draft
7 April 95	0003.draft0.4	EIDR Draft
October 95	0003.draft0.5	Updated Draft
February 96	0003.1	Initial release
22 June 98	0012.1	Delivery to ESA

D Acknowledgement

We acknowledge Phil Smith at MSSL for his contribution to Section 3.

1 Introduction

1.1 Purpose

The purpose of this document is to satisfy the requirements of ESA following the ESA Software Engineering Standards (ESA PSS-05-0) and the Guide to the Software Architectural Design Phase (ESA PSS-05-04). The ESA specified request for this document which also states the purpose of this document is "Experiment on-board software: functional description including software task definitions, purpose, actions performed, inputs and outputs for each task, task control and scheduling information, synchronisation information and software flow diagrams" (Section 5.X.8.1 in Doc No. PS-RS-0028). This document describes the architectural design of the ^{XMM}OM DPU flight software.

This document supersedes the document, "Overview of the XMM Optical Monitor Digital Processing Unit (XMM-OM/PENN/TC/0026.01)." While this document describes an overview of the on-board software, the "DPU Software Detailed Design Document (XMM-OM/UCSB/SP/0004)," describes its detailed design, which will be provided in future.

1.2 Scope

This document is a description of the framework for the software development of the on-spacecraft or "flight" software for the Digital Processing Unit (DPU) for the XMM Optical Monitor. This includes both the operating system software and the scientific processing software running on the DPU. The Instrument Control Unit (ICU) software and ground support software are discussed elsewhere.

Software produced for the DPU consists of three software packages: 1) White software for the White DSP; 2) Blue software for the two Blue DSPs; and 3) Red software for the Red DSP. The White and Red software are further sub-divided into the "Science" software and the "Operating System." This entire suite of software, known collectively as the DPU Software, is covered by this document.

Some of the software described here can be run in simulation on a UNIX platform. For details on running simulations, see the documents "DPU Simulation User's Manual" (XMM-OM/PENN/ML/0002) and "DPU Electronic Ground Support Environment and Software Development Environment" (XMM-OM/PENN/SP/0005).

1.3 Font Conventions in this Document

Running text shall appear in this normal Roman font. *Italic font* will indicate special emphasis. Items that appear in the DPU Lexicon (Appendix B) will appear in a *slanted font*.

The filenames of source modules (C or assembly code, shell scripts, etc.), the names of units (e.g., subroutines) within modules, and variable names within code modules will be rendered in `"typewriter" font`.

In the sections that describe the code, the names of major *tasks* of the White and Red DSP science codes (within `whitedsp.c` and `reddsp.c`) will appear in LARGE CAPS. In the description of the White science code, the names of the *swap units* will appear in SMALL CAPS. For example, the *swap unit* DELIVERDATA is comprised of several functions, including its top level `deliverdata()`, in the source module `su_deliverdata.c`. DELIVERDATA is used in five of the White DSP *tasks*, including FINISH_FRAME and ENGINEERING.

1.4 References

1.4.1 XMM-OM Controlled Documents

Document Number	Title
XMM-OM/MSSL/ML/0005	XMM-OM User Manual Part 1A - Experiment On-Board Software - ICU
XMM-OM/MSSL/ML/0011	XMM-OM User Manual Part 3
XMM-OM/MSSL/SP/0007	^{XMM}OM Electrical Interface Specification
XMM-OM/MSSL/SP/0014	XMM-OM User Manual Part 4
XMM-OM/MSSL/SP/0056	Blue Detector Electronics Detailed Design
XMM-OM/MSSL/TC/0015	Fiber Taper Distortion Associated with the MIC
XMM-OM/MSSL/TC/0032	Blue Detector Engineering Setup Modes
XMM-OM/PENN/ML/0002	^{XMM}OM DPU Simulation User's Guide
XMM-OM/PENN/SL/0008	^{XMM}OM DPU Team C Language Coding Conventions
XMM-OM/PENN/SP/0005	^{XMM}OM DPU Electronic Ground Support Equipment and Software Development Environment
XMM-OM/PENN/TC/0004	DPU Processing for ^{XMM}OM
XMM-OM/PENN/TC/0010	The Effects of Differential Non-linearity in a Photon-Counting Imager and a Solution
XMM-OM/PENN/TC/0021	Absolute Pointing Determination for the ^{XMM}OM Detector

1.4.2 Other References

Motorola 1990, DSP56000/DSP56001 Digital Signal Processor User's Manual (DSP56000UM/AD REV 2).

Hatley, D.J. & Pirbhai, I.A. 1988, Strategies for Real-Time System Specification, New York: Dorset House.

Kernighan, B.W. & Ritchie, D.M. 1988, The C Programming Language, 2nd ed., London: Prentice Hall.

1.5 Document Overview

Section 2 is a description of the DPU and the tasks it is required to perform, and §3 describes the DPU's interfaces to related systems. A description of the system design methods, and the software architecture data flow diagrams and state transition diagrams is in §4. Descriptions of the software components (here each software file is a component) are separated into three sections: the master processor, White DSP software, the Red DSP Software and the Blue DSP Software in §5. Library header files of the DPU software are described in §6. Appendix A lists the acronyms and abbreviations used in this document, Appendix B is a lexicon of terms used in describing the DPU operation, and Appendix C contains a description of the compiling utilities and procedures used to compile the flight software and prepare it for loading into the DPU memory. This ADD is organized following the template laid out in ESA PSS-05-04.

2 DPU Overview

The DPU is responsible for on-board data handling and compression to conform with the telemetry constraint while maintaining flexibility and maximizing scientific performances. The DPU software will perform the following major scientific tasks:

- Data accumulation, aquisition of images, and fast mode processing
- Data processing
- Absolute pointing acquisition
- High accuracy spacecraft drift tracking
- Collection and transmission of science data

The DPU software will perform the on-board processing of the science data from the MIC detectors, and pass these data to the ICU for transmission to the OBDH. This on-board processing includes the filtering of data spatially and temporally in order to reduce the OM telemetry rate to the limit allowed by the spacecraft. The DPU also calculates the spacecraft drift from frame-to-frame using stars within the field of view.

Two major data collection modes are supported by the DPU. They are:

- Imaging a large detector area over a long (> 1 ks) exposure time (image mode)
- Collecting high resolution time series data over a small detector area (fast mode)

Despite the fact that the Blue detectors involve CCDs, the detectors are fundamentally different, and the format of the data received by the DPU reflects that fundamental difference. The detector is a Micro-channel plate Intensified CCD (MIC). It is read out as a "photon counting" device. The MIC CCD is read out continuously (with a typical CCD frame time of 10 ms) and the Detector Electronics process the image by centroiding the "photon splashes" induced on a phosphorous screen by electron clouds emitted from the micro-channel plate. The data transmitted to the DPU are in "event" format – each word represents a photon detected, and contains the location on the detector of the event.

The data rate from the MIC detector is dependent on the number of pixels in the exposure and the flux from the source – the more photons are received, there are more events to be transmitted to the DPU.

It will not be possible to telemeter the information obtained from every photon detected owing to telemetry rate limitations. The DPU therefore must reduce the data before telemetering. This is done by filtering the data spatially and temporally – by restricting the data telemetered to user-defined "windows" (rather than the entire detector collecting area) and accumulating, or "integrating" images over time. Because of drifts in the spacecraft attitude, objects will wander on the focal plane at the resolution of the OM, and the accumulation of image data in the DPU memory must be accompanied by an algorithm to co-add windows from a series of short ($\sim$10 s) *tracking frames*, the windows having been shifted by the amount of the drift measured with respect to a *reference frame*.

When the telescope slews to a new position, an *acquisition frame* lasting one *frame time* ($\sim$10 s) is taken with the Blue detector, in full-format, low-resolution mode, normally behind a V filter. A catalog is made of the brightest stars (512 maximum) in the field of view. As many as 32 objects from the HST Guide Star Catalog (or similarly bright stars with accurately known sky positions), whose coordinates are up-linked and delivered to the DPU by command, serve as *absolute guide stars* for field acquisition. A pattern-matching algorithm is employed to match the *absolute guide stars* with the cataloged bright stars. The search space is limited to the estimated pointing uncertainty of $\sim 1'$. Best fit values are calculated for the absolute pointing errors ΔX, ΔY, and $\Delta\Theta$. These errors give the discrepancy between the commanded bore-sight orientation and that obtained and are used to determine which detector pixels correspond to the observer-selected *science windows*.

A similar sequence of events precedes each *exposure*. First, a *reference frame*, lasting one *frame time*, is exposed, again with the Blue detector in full-format, low-resolution mode, this time using the observer-selected filter. Then, as above, a catalog is made of the brightest stars in the field of view. Up to 10 of the cataloged stars which meet several acceptance criteria (*e.g.,* range of field position, morphology, etc.) are selected to be used as *guide stars*. *Science windows* in addition to those specified by the observer are established automatically by the DPU around the *guide stars*; these are sometimes referred to as "tracking windows."

A set of non-overlapping data collection windows called *memory windows* contain the *science windows*. A *memory window* corresponds to a fixed sample of detector pixels whereas a *science window* corresponds to a fixed patch of sky. Thus *science windows* may drift around within *memory windows* as the satellite drifts, however the latter are intended to be sufficiently large that such drift should not cause the edge of a *science window* to cross the boundary of the *memory window* in which it resides. Each *memory window* is comprised of one or more *detector windows*.

It is envisioned that an interactive, user-friendly (and thus far non-existent), ground-based program will enable an observer to specify the parameters which collectively characterize a set of *science windows* and a corresponding (optimized) set of *memory windows*. The parameters will be up-linked and passed to the DPU by command. The DPU will *(a)* allocate, as necessary, additional *memory windows* to cover the *tracking windows*, and *(b)* assign *detector windows* to the *memory windows*. On command, the *detector window* parameters will be passed to the ICU, which will configure the MIC detector to provide the proper data format. Within the DPU, these parameters will be passed (via *global memory*) from the White DSP, where they are calculated, to the Blue DSPs, which use them to convert blue detector data to memory addresses.

An *exposure* consists of an integral number of *tracking frames* during each of which the satellite is supposed not to have drifted significantly relative to the $0.''5$ extent of a MIC detector pixel. The *guide stars* chosen from the *reference frame* are relocated in each *tracking frame* and best fit values for the drift in x and y (and the roll) are calculated. The roll, which should be small, is ignored, and the translation, rounded to the nearest pixel, is used in the Fast Mode and Image Mode algorithms to shift the *science windows* acquired during the *tracking frame* into alignment with the *reference frame*, thereby maintaining spatial registration.

The ICU dictates which of its major functions — field acquisition, guide star selection/window configuration, or tracking/data accumulation - the DPU performs as if it were submitting batch jobs to a processor. The ICU is notified when these various functions have been completed. All the timing is handled by the DPU. For example, the enabling and disabling of detector event processing which correspond, respectively, to the beginning and end of a *reference frame* exposure, or a 1000 s image exposure, are controlled internally; the White DSP, which keeps an eye on the DPU bus cycle (millisecond) count, sends signals to the Blue DSPs at the appropriate times to initiate or halt the flow of data into *global memory*. Data are queued for delivery to the ICU at appropriate intervals and the ICU is given an "alert" whenever new data are introduced to the queue. A typical operational sequence is described in §5. Explanatory notes intended to facilitate understanding of the ICU control of the DPU are provided in that section.

2.1 Hardware Overview

The major hardware components of the DPU is shown in Figure 1. A large block of *global memory* (4 M 16-bit words plus 1 M 24-bit words) stores temporary and accumulating data. Four processor cards access the memory in series via a global bus, with access to the bus managed by an arbiter board. Each processor is granted access to the bus once per millisecond. The DPU shares the Digital Electronics Module crate with the ICU and an integral power supply that supplies power to the DPU and ICU.

The four Motorola 56001 Digital Signal Processors (DSPs) have *local* (*on-chip* and *on-board*) *memory*, plus access via the global data bus, to *global memory*. Each of the four processors is assigned specific tasks and are labeled according to the tasks that they perform: White DSP, Blue DSP 1, Blue DSP 2, and Red DSP whose details are described in the next section. The White processor is responsible for overall management of the other processors, communication with the ICU, initial field acquisition, and spacecraft drift tracking. The Blue DSPs are responsible for data collection and initial processing of the Blue Detector data. The Red DSP is responsible for performing the *shift-and-add* calculation.

2.1.1 Global Memory

The **DPU** *global memory* consists of 12.5 Mbytes of RAM, and is accessed via the global memory bus. If a processor does not release the global memory bus, after several bus cycles the watchdog on the arbiter card will reset the DPU. After a reset, the DPU will return to the Boot–Idle state. Each DSP card has its own *local memory* which can be accessed only by the DSP on the card. *On-board memory*, consisting of 32 words of 24 bit words is available to each processor for program and variable storage. Each processor also has an *on-processor memory*. The *global memory* is divided into three partitions, *Small Word Memory* (SWM), *Big Word Memory* (BWM), and *Program memory*. The four DSPs share the *global memory* via a global data bus and global address bus. The global RAM map can be found in p22 XMM-OM User Manual Part 1A - Experiment On-Board Software - ICU (XMM-OM/MSSL/ML/0005.1).

- *Program Memory (Storage)* consists of 0.5 Mwords of RAM arranged in 24 bit words, and 8 Kwords of PROM also arranged in 24 bit words.
- *Small Word Memory* consists of 4 Mwords of RAM arranged in 16 bit words. It is used primarily for temporary storage of tracking frame data. Data for the engineering modes and full frame applications (*e.g., acquisition* and *reference frames*) will also be stored there.
- *Big Word Memory* consists of 1 Mword of RAM arranged in 24 bit words. It will be used for storage of the current and previous *accumulated images*, as well as processed engineering mode data (required only for centroiding confirmation mode).
- *PROC memory* is a 32768 word section of BWM used for storage of parameters for inter-DSP communications.
- *Program memory* consists of 0.5 Mwords of RAM arranged in 24 bit words, and 8 kwords of PROM also arranged in 24 bit words.
- *Read Only memory* (ROM) contains instructions to boot the computer, while Ramdom Access Memory (RAM) has no previous memory, often random values or zeroes.

The PROM contains the bootstrap software for all four processors for loading the software from the program storage into the processor's *local memory*. The program storage, containing the operating system and the data processing software is keep-alive and can be modified during the flight.

The *swap units* are designed to use the bus sparingly and to enable as much number-crunching as possible between successive bus access opportunities. It is estimated that the DSPs will be able to read/write 256 words from/to *global memory* per 1 ms DPU bus cycle; this rate is given by the static variable `WORDS_PER_BUS_ACCESS`.

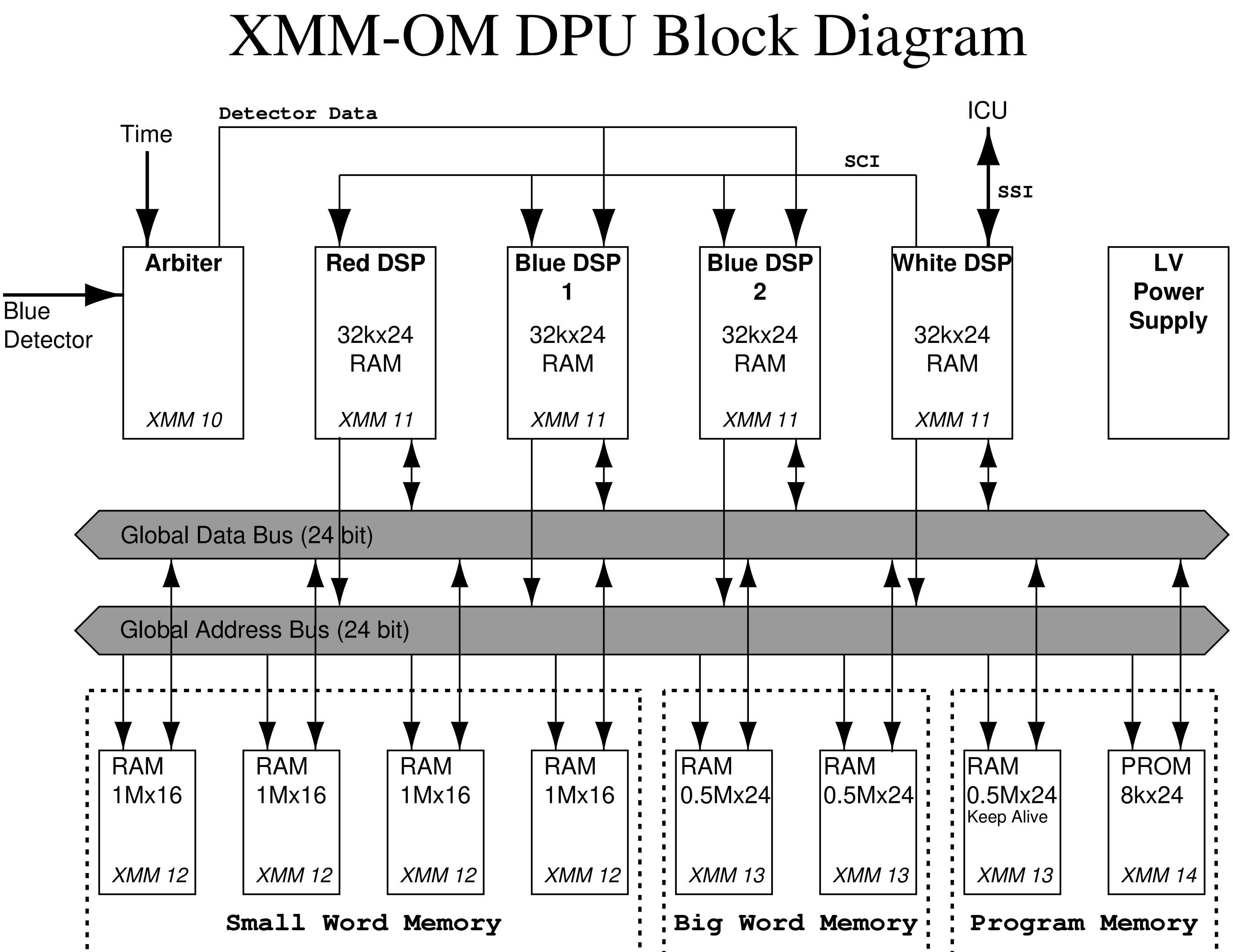

Figure 1: Block diagram of the XMM OM DPU.

2.2 DPU Software Overview

DPU is composed of three processors: White DSP, Red DSP and Blue DSP. While White DSP is the master processor, Blue DSP converts the event data into an image format data, and processes fast mode data, and Red DSP performs the shift-add operation. The DPU software overview diagram in Figure 2 illustrates the main software components in the three processors, and the data processing tasks, and the connection to ICU and BPE/DPU data capture interface. The hardware paths are such as Serial Communcatons Interface (SCI) for 'real-time' commanding from the White DSP to the Red and Blue DSPs. There is 'commanding' or 'configuring' by White DSP to Red or Blue DSPs via the PROC area. No in/out arrows are attached to the PROC area, because almost every software components uses PROC.

2.2.1 White DSP

The White DSP is the master processor, which handles all DPU/ICU communications. The White DSP orchestrates the activities of the other processors and handles most of the data reduction performed by the DPU.

- The White DSP:
  - Commands and controls the other DSPs
  - Communicates with the ICU and performs DPU–ICU I/O functions (receives commands and transmits data)
  - Manages observing configurations
  - Performs pointing acquisition
  - Calculates the spacecraft drift (tracking)
- The White DSP runs an operating system (DPUOS) written in assembly language and C that supports the scientific software written in C.
- The scientific data processing *tasks* are managed by a central code which invokes major code fragments (the "*swap units*") to perform the *tasks*.
- All *swap units* are stored in the *global memory* (RAM). Due to limited space in *local memory*, only the active *swap unit* can be stored in White DSP *local memory*. The *swap units* are loaded into the local RAM through the DPUOS.

2.2.2 Blue DSP 1 & Blue DSP 2

Data from the Blue Detector are received in "event format", Blue Detector event records representing photon locations (centroids of photo-events). The primary task of the Blue DSPs is to convert the "event format" data, which is a list of events, into an "image format" data, or an image stored in memory. The Blue DSPs form an image in the DPU's *global memory* by calculating and incrementing the address in *global memory* of each event received – there is a one-to-one mapping of the coordinates of the event received to a location in *global memory*.

The Blue DSP software is contained within one component, `blue.asm`, which is identical in both Blue DSPs. It is coded in assembly language owing to the fast computations required by the expected event rate. The address calculation algorithm is a simple calculation which is unlikely to be modified in the lifetime of the mission. The processing parameters, such as the location in *global memory* of the accumulation image, are controlled by the White DSP, and are updated for each exposure.

- Two Blue DSPs are necessary to handle the expected event rate.

- The DPU arbiter can be commanded to route all the Blue Detector events to one or the other Blue DSP, or alternate the events between the two Blue DSPs.
- The Blue DSPs accumulate the image in memory for one *tracking frame time* (between 10 and 20 seconds).
- Blue Fast Mode processing is performed by the Blue DSPs.

The Blue DSPs are responsible for blue detector event processing and the Blue Fast Mode.

As mentioned in §2, blue detector data are delivered alternately to the Blue1 and Blue2 DSPs. The raw data are buffered and the *global memory* is updated when the global bus becomes available. Whenever a Blue DSP is not preoccupied by buffering incoming data or updating the *global memory*, it is busy turning the raw data into memory addresses.

Occasionally, an observer-specified frequency could not exceed 500 Hz, or once per 2 ms, when the bus becomes available. Fast Mode data are extracted from *global memory*, processed into the proper format (see Fig. 10), and saved for later delivery in the telemetry stream. All of the Fast Mode windows handled by a given Blue DSP will be sampled at the same rate, however, at no cost, the windows handled by Blue1 may be sampled at a different rate than those handled by Blue2.

2.2.3 Red DSP

The Red DSP is used for only one purpose - performing the *shift-and-add* operation on each *science window* of each *tracking frame* of the science *exposures*.

- The Red DSP has a "basic" operating system to support high-level (C) software.
- In nominal flight operation, the Red DSP runs in an infinite loop, waiting for the White DSP to command it to perform one of its two *tasks* (RED_INITIALIZE just after loading DPUOS, and RED_ACCUMULATE_IMAGE during science observations).
- Red DSP receives commands from White DSP via SCI link, and will share required parameters (*e.g.,* status flags) with White DSP via the *PROC area* of *global memory*.

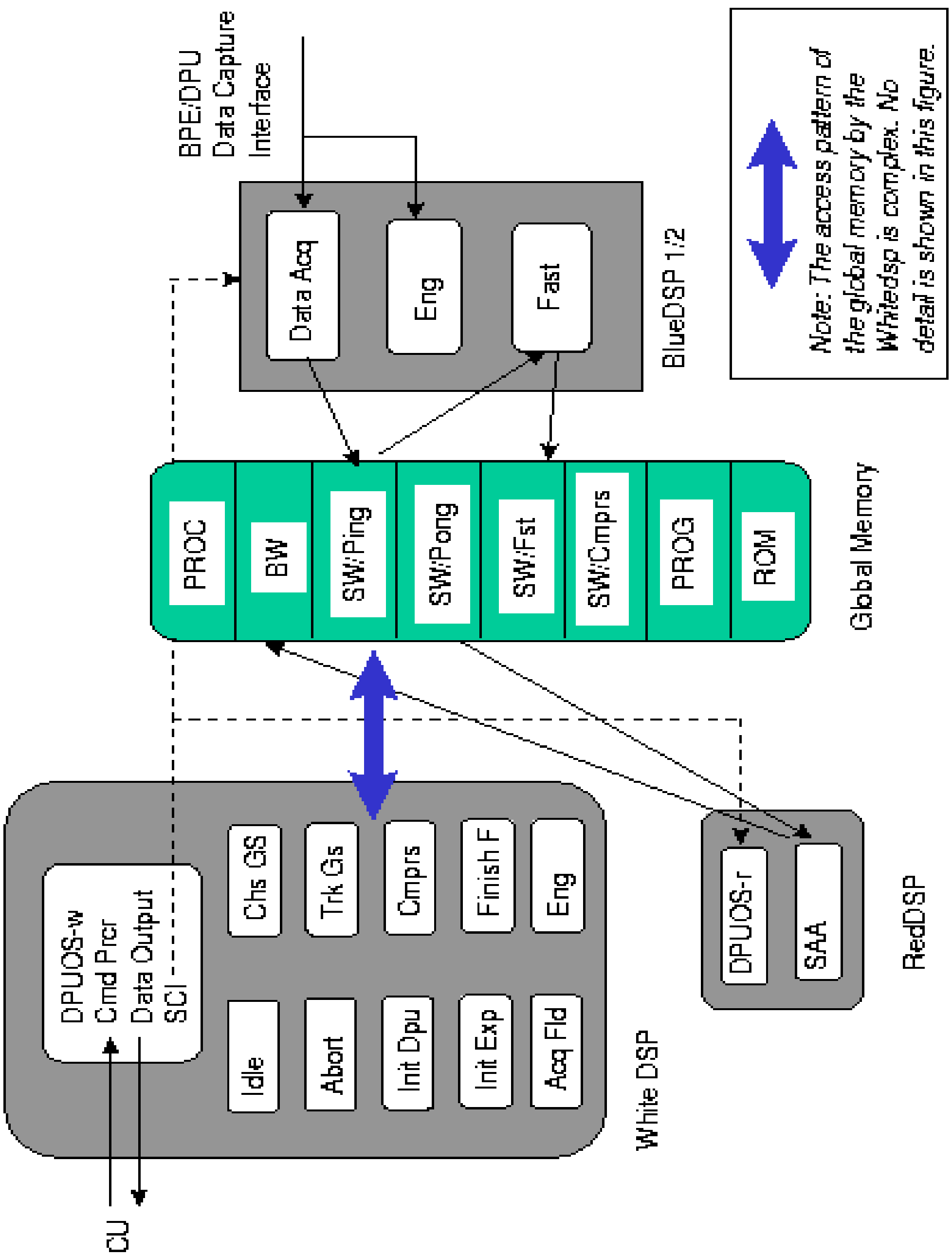

Figure 2: DPU Software Overview. The thick arrow shows a simple presentation of task execution between White DSP and Global Memory. The solid and dotted lines indicate data flows, and dedicated hardware path, respectively. For details, see the text.

2.3 DPU States and Commands

The DPU has three major states of process: the first state is execute "Boot–Idle", the second state is "DPUOS-Idle", and the third state is "Executing Task" as shown in Figure 3, the state transition diagram for the DPU. When one starts to run an exposure, these will be the steps noticed as "Fred, Jim, and executing task". In this section, the three major states are described, and the commands for performing the state are listed one by one in execution order. The records of those commands can be found in "XMM-OM User Manual Part 4," project document XMM-OM/MSSL/SP/0014. Figure 3 is the state transition diagram for the DPU. Once the spacecraft powers up, Keep Alive Line (KAL) is turned on, i.e. the power will be on into DPU, DPU memory will be loaded. When in the boot-idle state, the spacecraft will load the ICU and be waiting for loading the DPU. After loading the DPU, DPUOS-idle states is in the next command.

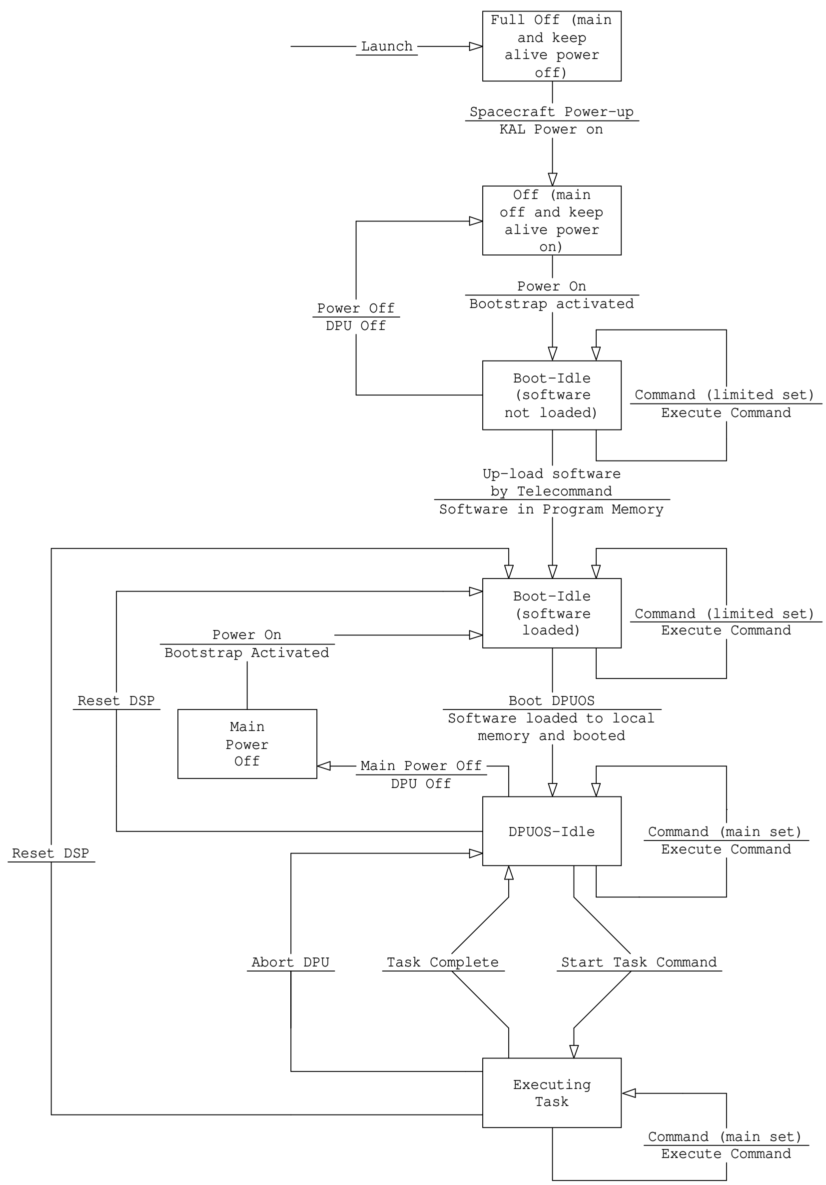

Figure 3: : DPU State Transition Diagram. The DPU has three major states: 1) Boot–Idle; 2) DPUOS-Idle; and 3) Executing Task. These states are discussed in detail in the text.

2.3.1 Boot–Idle State and its Commands

When the DPU is initially powered on, the White DSP initiates its Boot–Idle state. The commands recognized *only* in Boot–Idle are listed below with a short description in the parentheses. The command name is based on the key words of the description.

- `IC_ENBL_LOAD_CODE` (lock/unlock loading of code)
- `IC_LOAD_CODE` (load code into RAM)
- `IC_LOAD_DPUOS` (instructs DPU to load the DPUOS. This is used when the DPU is in Boot Ready state)
- `IC_SEL_RAM_BANK` (specifies the bank of program RAM to be loaded)

2.3.2 Boot–Idle and DPUOS–Idle Commands

Commands recognized both in Boot–Idle and DPUOS–Idle (see below) are:

- `IC_DUMP_RAM` (dump n 24 bit words from global RAM and then zero the RAM)
- `IC_RESET_DSP` (hardware reset of the selected DSP)

2.3.3 DPUOS–Idle States

The DPUOS–Idle state is entered in response to `IC_LOAD_DPUOS`. At that point, the White DSP traffic control routine (`whitedsp()`) is in its IDLE *task* loop, waiting to respond to a command to start a major *task* (The White DSP IDLE *task* is equivalent to the DPUOS–Idle state.) In this state all commands from the ICU, except those recognized in the Boot–Idle state only (see above), are recognized. The current list is:

- `IC_RESET_DSP`
- `IC_SEL_KLINGON` (select which DSP among blue1, blue 2, red, and white, will be the klingon)
- `IC_ENBL_DSP` (enable/disable DSP)
- `IC_ENBL_EVENTS` (enable/disable data generation from DSP)
- `IC_DIRECT_DATA` (direct detector data to either blue1, blue2, or both blue DSPs)
- `IC_SEND_SCI_CMD` (sends the command "message" from the white DSP to the targe DSP over the SCI interface)
- `IC_REQ_DATA` (request 'number' blocks of data)
- `IC_FLUSH_QUEUE` (request for the white processor to dump all data currently queued for output)
- `IC_DUMP_LOCAL_RAM` (request a dump of local RAM - response in DR_LRM)
- `IC_LOAD_LOCAL_RAM` (load white local RAM)
- `IC_DUMP_PROG_RAM` (requests a dump of program RAM - reponse in DR_PROG_DUMP)
- `IC_DUMP_RAM`
- `IC_DUMP_RAM_N_ZERO` (dump n 24 bits words from global ram for an acceptance / confidence test mode of operation, and then zero the RAM)
- `IC_LOAD_GLOBAL_RAM` (load global RAM)
- `IC_REPORT_TRK` (report tracking history -causes DA_TRK alerts to be sent)

- IC_REPORT_DIAGS (activate/deactivate log- keeping for swap unit "module")
- IC_SET_FRAME_TIME (set tracking frame duration, where 1 DPU cycle = 0.001 seconds)
- IC_SET_EXP_TIME (define exposure duration as some integral number of tracking frames)
- IC_SET_EXP_ID (specify exposure ID)
- IC_CONF_GS_SEL (set number of guide stars and their selection criteria)
- IC_LOAD_REF_STARS (load list of field acquisition references stars)
- IC_LOAD_FILT_CONF (inform DPU of type of filter being used)
- IC_LOAD_MEM_WDW (uplink memory window configuration data)
- IC_LOAD_SCI_WDW (uplink science window configuration data)
- IC_ENBL_VERBOSE (enables long form of data)

2.3.4 White DSP task Commands

The commands which invoke major White DSP *tasks* by changing the value of the White DSP `task_id` variable are:

- IC_INIT_DPU (zeroes memory, readies swap units)
- IC_INIT_EXP (invokes the INIT_EXP task. This sets up the DPU to acquire detector data in 1K * 1K format, i.e. detector binned by 2)
- IC_ACQUIRE_FLD (command DPU to excute a task for acquiring a field)
- IC_CHOOSE_GS (command DPU to execute a task for choosing guide stars)
- IC_TRACK_GS (command DPU to execute a task for tracking guide stars)
- IC_ABORT_DPU (command DPU to abort - it leaves an exposure without disturbing the DPU by initiating the abort task)
- IC_ENBL_ENG (enable/disable DPU engineering mode data)

2.3.5 Executing Task

Several of the commands recognized in DPUOS–Idle are used to initiate major White DSP *tasks*, as indicated above. The action taken upon *task* completion varies from one *task* to another. The *tasks* INIT_EXP, INITIALIZE, ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, and ENGINEERING check the value of the `task_id` variable. If it is unchanged during *task* execution, it is set to IDLE, thus returning to the DPUOS–Idle state. If `task_id` has changed, presumably by command from the ICU, control returns to the `whitedsp()` control loop and the commanded *task* is invoked. The TRACK_GUIDE_STARS *task* also checks `task_id`, but sets it to COMPRESS_DATA rather than IDLE if it has not changed during execution. COMPRESS_DATA sets `task_id` to FINISH_FRAME upon completion during normal tracking operations. FINISH_FRAME checks the value of `task_id`. If it has changed, it is reset to ABORT (TBC). If it has not changed during execution, it is reset to TRACK_GUIDE_STARS if the exposure is not complete, and to IDLE if the exposure is complete. The FLUSH_COMPRESS *task* simply sets `task_id` to IDLE when it finishes, thus returning to the DPUOS–Idle state.

Barring interruption by command from the ICU, the IC_TRACK_GS command initiates a sequence of TRACK_GUIDE_STARS–COMPRESS_DATA–FINISH_FRAME executions, as indicated above. The number of iterations is the number of *tracking frames* in the *exposure*. These *tasks* invoke each other in the proper sequence until the *exposure* has finished. The latter two *tasks* are not commandable by the ICU.

Under normal circumstances, one exposure starts, its completion is controlled by the DPU with a progress report sent regularly by the ICU. If the DPU receives an ICU command, the current exposure may be corrupted. During execution of any *task*, the ICU may still send commands to the DPU. Behavior will depend on the specific command.

2.3.6 Red DSP States

The Red DSP has states analogous to DPUOS–Idle and Executing Task. The Red DSP starts up into its "RedOS–Idle" state, which is an infinite loop in which the value of `red_task_id` is examined. When that changes from `RED_IDLE` to another value (via SCI link command from the White DSP) the indicated Red DSP *task* is executed, i.e., the Red DSP enters an "Executing Red Task" state. All three functional Red DSP *tasks*, RED_INITIALIZE, RED_ACCUMULATE, and RED_ABORT, set `red_task_id` to `RED_IDLE` upon completion, thus returning the Red DSP to its "RedOS–Idle" state. The Red DSP *tasks* are not interruptible.

2.3.7 Blue DSP States

Because there is no operating system, the details of states are described in section 4.3.

2.4 DPU Modes

There are several modes of DPU operation for taking *exposures*. For normal science observations, the Blue detector system allows us to use an "image" mode and a "fast" mode simultaneously. Each mode will have its own set of *science windows*, which correspond to fixed patches of sky. The *science windows* will be embedded in *memory windows*, which have fixed locations on the detector. Thus, *science windows* will move around in their *memory windows* as the spacecraft drifts during an *exposure*. During science observations, data on the *guide stars* used for *tracking* will be kept in *tracking science windows*. More than one *science window* may exist within a *memory window*, and *science windows* may overlap.

To clarify this a little further: In the case of an *image mode science window* embedded in another, both share the same *memory window*. In the ping/pong areas in *small word memory*, the information in the pixels used by both *science windows* is stored only once. On the other hand, the *shift-and-add* process loops over all *image mode science windows* separately, so in the *accumulated images* in *big word memory* the duplicated pixels are stored twice.

2.4.1 Image Mode

In image mode, photons will be accumulated for the duration of an *exposure*. *Image mode science windows* in the series of 10 s *tracking frame* images will be subject to the *shift-and-add* process to account for spacecraft drift during the *exposure*. These re-registered data will be summed into an accumulating image, such that the absolute position in sky coordinates is maintained for every photon. The raw images may contain as many as 10^6 photons in any pixel. At the end of an *exposure*, the *accumulated image* data for the *image mode science windows* are compressed and passed to the ICU for down-linking.

2.4.2 Fast Mode

It is possible to select up to two small regions of the field of view for higher time resolution observation. Such regions will have *fast mode science windows*, each of which may include up to 512 *detector pixels*. At an observer-specified interval (the *fast mode slice period*), these windows will be checked to determine the number of counts detected in each pixel of the window. The DPU-imposed constraint on the Fast Mode sampling time is $\sim$ 100 ms; detector readout or other constraints may be more restrictive, however.

The *fast mode slice period* must be an integer number of MIC *CCD frame times*. In general, this will *not* be an integer factor of the *tracking frame time*. To avoid possible beating effects, fast mode will synchronize to CCD frame markers supplied by the MIC detector.

The MIC CCD is continuously read-out. Once a CCD frame has been read out of the frame storage area of the CCD, the next frame is transferred into the frame storage area. The time required to read-out a CCD frame, and thus the *CCD frame time*, is dependent on the total number of CCD pixels in the *science windows*. A CCD frame time will typically be about 10 ms, but can range between 2 ms and 11 ms (TBC). Inside the DPU, the BPE decides the CCD frame time. Because of this mode of detector operation, it is possible to have both *fast* and *image mode science windows* active at the same time. Thus, an observer might specify a large *image mode science window*, to cover, say, a star forming cloud, and a couple of *fast mode science windows* to record the short timescale variability (e.g., due to flares) of two known T Tauri stars in the cloud.

Because the *tracking frame time* and *fast mode slice period* in general will not be the same, no spacecraft drift corrections will be applied to *fast mode* data. Therefore, since *fast mode* windows must be small, they must be specified accurately.

2.5 Engineering Modes and Data

There will be six types of engineering mode that will be required from time to time during the mission and seven types of engineering data (mode 3 creates the sub-type data of 3 and 7). For these engineering modes, the ICU will command the Blue detector to transmit the desired type of engineering data to the Blue DSPs. On the DPU side, software window setup will not correspond to the detector area(s) from which data are to be saved. Rather, window setup will serve only for DPU memory management. The data format bit map for each mode is illustrated in section 2.2.6.2.4 of "XMM-OM User Manual Part 1A - Experiment On-Board Software - ICU" (XMM-OM/MSSL/ML/0005.1). For further details see "Blue detector Engineering Setup Modes" (XMM-OM/MSSL/TC/0032.01).

2.5.1 Raw Data (Mode 0, 1, and 2)

The Blue detector will transmit raw data event records; no address calculation will be performed by the DPU software. The data will be limited by available memory and possibly by an "exposure" time. Mode 0, 1, and 2 are for raw data with both blue DSPs, only blue DSP 1, and only blue DSP 2, respectively. The raw data for mode 0 has an output file of a concatenation of BLUE1 and BLUE2 because, when both blue 1 and 2 are on, events are sent even-odd to each sequentially as divided by the arbiter card. This is done by the hardware.

2.5.2 Centroid LUT Calculation (Mode 3)

Centroiding is the process of locating the position of an event to an accuracy greater than that of a CCD pixel which is divided into 8×8 bins of uneven size, otherwise known as sub-pixels. In an ideal case equally spaced boundaries among these sub-pixels are expected; for a real image intensifier and CCD camera, a correction for the centroiding can be made by using a look-up table RAM. For each event and each x and y axis, the processing electronics produces two 8 bit numbers, labelled by M, and N (the input is an M,N image which is obtained by an electronics). The ranges of M and N are from -128 to 127 and from 0 to 255, respectively. The ratio of M and N infers a fractional position within a CCD pixel of the event, in other words, boundaries of the sub-pixels. The boundary numbers between the eight subpixels are calculated using look-up tables (LUTs), which are two tables containing all possible results of the division: one for X axis and the other for Y axis. These tables are values of M/N and their integrated count number calculated from

M, N images. The output data M,N images are two 256×256 arrays (this output is defined as sub-type data 7). Details may be found in "Blue Detector Engineering Setup Modes" (XMM-OM/MSSL/TC/0032.01) by D. A. Bone.

2.5.3 Full Frame at High Resolution (Mode 4)

In this mode the output is a full frame at high resolution (8 "detector" sub-pixels per CCD pixel of a 256x256 image yields a 2048×2048 pixel image). The output of this mode is a full science image which has been accumulated and boundary corrected (by the centroiding algorithm) using the boundary values supplied. This mode will require *all* of *small word memory*.

2.5.4 Centroid LUT Confirmation (Mode 5)

For this mode, we first obtain a high resolution full frame (repeat Mode 4). This frame is divided into 16 512×512 sectors, which will be considered separately in subsequent processing. That processing will sum every eighth pixel in X and Y in a sector to produce 8×8 grid images each with 8×8 pixels which represent the average distributions of counts in the detector sub-pixels of the CCD pixels of each sector. The image requires a small word memory. The input image is a high resolution full frame image and the output image is a pixel centroiding map; an input file of a flat image will create a flat centroiding confirmation map, and the centroiding confirmation map will be brighter where the image is brighter.

2.5.5 Event Height & Energy (Mode 6)

This mode will be used to characterize the intensifier and CCD characteristics periodically throughout the mission. The Blue detector will transmit two 256 element arrays containing histograms of event height (i.e., number of counts in each of 256 height). The histogram of counts for each 256 height can be used to examine the intensifier and CCD characteristics, which would be an important output for engineers.

2.6 Communication Between Processors

There are two methods for communicating information among the DPU's four Motorola 56001 Digital Signal Processors (DSPs).

The first is via the the Serial Communications Interface (SCI), which is a three-pin serial communication port built into the DSPs (see the DSP56000/DSP56001 Digital Signal Processor User's Manual). The SCI is a unidirectional commanding link used for time-critical commanding. Only the White DSP can send commands to the other DSPs. The two Blue DSPs and the Red DSP do not initiate communication on the SCI interface.

For regular information transfer, the DSPs communicate through the *global memory*. For example, the *PROC area* of *global memory* holds, among other information, status flags for several DPU functions, including the status of the *shift-and-add* on the Red DSP. When the TRACK_GUIDE_STARS *task* has finished calculating the spacecraft drift for the *current tracking frame*, it writes a few variables needed by the `reddsp` code into the *PROC area* of *global memory*, commands the Red DSP via the SCI link to begin the *shift-and-add*, and sets the *PROC area* variable `red_saa_status` to `SAA_START`. Then the COMPRESS_DATA *task* is run. It works on compression until a short time (`COMPRESS_TIME_BUFFER` milliseconds) before the end of the current *tracking frame time*. Then it stays in a loop until `red_saa_status` has the value `SAA_DONE`. The `reddsp` code writes that value to the *PROC area* when it finishes the *shift-and-add*. In this way, the Red DSP informs the White DSP of its status. Also, many of the commands sent via the SCI tell the Red or Blue DSPs to read information from the *global memory*.

2.7 Heartbeats

There are two types of heartbeats, the SSI and the SCI heartbeats. The SSI (the interface between the DPU and the ICU) heartbeat is a signal sent from the DPU to the ICU saying, "We are here!" It is sent every 10 seconds (time interval set during compilation) whenever the DPU is powered, whether it is in Boot-Idle state, DPUOS–Idle state, or Executing Task state (cf. Figure 3).

The SCI (interface between the DSPs) heartbeat is a signal sent by the Blue 1, Blue 2, and Red DSPs to the White DSP indicating that they are operational. Every 1.0 seconds (time interval set during compilation), the White DSP asks each of the other DSPs, in turn via SCI, if it is alive. Each must respond with the "I am alive" signal written to *global memory* (when that DSP has access to the global address bus), which the White DSP checks the following bus cycle. If White does not read the "I am alive" signal from *global memory*, it issues an alert on the SSI to the ICU (`DA_SCI_HBEAT_ERROR`).

2.8 DPU Programming Idiosyncrasies

There are a few issues related to the actual programming that deserve brief mention here. These do not have to do with coding standards, *i.e.,* the format or appearance of the code, which is described in "^{XMM}OM DPU Team C Language Coding Conventions" (document XMM-OM/PENN/SL/0008.02).

The `switch/case` constructions will be relatively inefficient on the DPU. A faster alternative is to use equivalent `if`, `else if` constructions.

Division is a relatively slow operation, and should be avoided when possible. For example, we multiply by `0.001` rather than divide by `1000.`

In two dimensional arrays, note that the second (Y) index varies faster than the first (X) index (see Kernighan & Ritchie p. 217).

3 System Context: DPU Interfaces

There will be two DPUs on board ^{XMM}OM, one in each of the two (one primary, one redundant) Digital Electronics Modules (DEM [a.k.a. OM2]). Only one DPU will be operational at a time. Each DPU has a bi-directional interface to the ICU, a uni-directional time interface, and uni-directional interfaces to either the prime or the redundant MIC detector.

The ^{XMM}OM electrical interfaces are defined in Electrical Interface Specification (XMM-OM/MSSL/SP/0007). For details on the DPU interfaces, refer to that document. There are a total of three interfaces between the DPU and the other subsystems; two are serial interfaces between the DPU and the ICU and MIC detectors, the third interface is a time interface.

3.1 The DPU – ICU Serial Synchronous Interface (SSI)

The ICU and DPU communicate via the bi-directional Serial Synchronous Interface (SSI), which is carried on the DEM backplane. The definition of the SSI is also in XMM-OM/MSSL/SP/0007 "Electrical Interfaces Specification". The commands sent by the ICU to the DPU, and the data types sent from the DPU to the ICU are defined in ICU – DPU Protocol Definitions (XMM-OM/MSSL/SP/0014). Those data types include various alert messages that inform the ICU of the status of DPU processing.

3.1.1 Hardware

Both the ICU and the DPU can send and receive data on this interface but the ICU is the master.

The interface consists of:

- SSI_CLK: a continuous clock signal generated by the ICU
- SSI_ENV_TX: active high when data present
- SSI_DATA_TX: 16-bit data
- SSI_ENV_RX: active high when data present
- SSI_DATA_RX: 16-bit data
- Signal return

Commands are sent from the ICU to the DPU. Science data is passed from the DPU to the ICU when demanded by the ICU. Alerts are sent (unrequested) by the DPU to the ICU. There is no direct feedback as part of the protocol and there is no error correction nor checksums. The interface can be thought of as the same irrespective of direction.

The SSI clock frequency is 125 kHz producing a period of 8 μs (1 bit-period). The SSI 16-bit data words are separated by at least one bit-period and at most the SSI block gap (defined in software). The SSI data blocks are separated by at least the SSI block gap (defined in software).

3.1.2 Transmitting data

The words that constitute the block are sent not more than the SSI block gap apart and, when finished, the software must wait for at least the SSI block gap before sending more data. The receiving software must wait for a little longer than the transmitting software's block gap to be sure to see the gap. A factor of two is sufficient.

3.1.3 Receiving data

The data being received must be read suitably fast and if the time between any two words is greater than the SSI block gap, the gap will be considered a block gap. All blocks contain a length as their second word so errors caused by an accidentally lengthened word gap may be identified (see data format).

3.1.4 SSI block gaps

Because the SSI block gaps are defined and used only in software they can be set to different values in different versions of the code and they can be different depending on the direction of the data (ICU⇒DPU or DPU⇒ICU).

SSI block gaps as defined by the ICU software

	EPROM code	Uploadable code
ICU ⇒ DPU	> 4 ms	> 4 ms
DPU ⇒ ICU	6 ms	4 ms

SSI block gaps as defined by the DPU software

	EPROM code	Uploadable code
ICU ⇒ DPU	2 +/- 1 ms	2 +/- 1 ms
DPU ⇒ ICU	15 +/- 1 ms	15 +/- 1 ms

The ICU's SSI hardware will give an interrupt (used by the ICU's software) at the end of the first word of each block. The ICU software must then read this first word before the end of the second word. The time for this is 16 bit-periods for the word and a minimum of 1 bit-period for the word gap. So the software must be able to respond to the interrupt and read the word in 136 μs.

An overflow (OVF) bit in the hardware SSI status word is made active (low) if a data word is not read before the arrival of another.

3.1.5 SSI errors

If the DPU resets whilst transmitting the first part of a word, that word will be truncated and the envelope will be truncated resulting in an earlier than expected "data receive" flag which will not be able to be processed in time and cause an overflow on the ICU.

If the DPU resets whilst transmitting the last part of a word, that word and the envelope will be truncated but not so much that the ICU's software cannot keep up as in the previous case. This will result in a corrupt last word and, except in the case of a reset during the last word, a truncated SSI block. This will be detected and handled properly by the ICU's software.

3.1.6 Data format

The data format is described in XMM-OM ICU-DPU Protocol Definitions. Each SSI data block consists of

1. 16-bit type - the block type
2. 16-bit length - the number of 16-bit words following this word (i.e. total length - 2)
3. the rest of the data

The data types are grouped into categories as follows:

- Regular DPU to ICU data blocks: Regular science data.

- DPU priority data: These contain science data that is sent out as soon as it is available rather than at the end of an exposure.
- DPU RAM dumps: RAM dumps.
- DPU to ICU alerts: Alerts from the DPU to signify something is has happened, is ready or an error has occured.
- ICU to DPU commands: Commands to the DPU

3.1.7 Further detail on the DPU software

The DSP converts a serial SSI word to parallel word. Each received word generates an interrupt. The SSI ISR pushes the word into a circular buffer. The 1ms ISR checks the COLLECTING_A_COMMAND bit. If it is zero (cleared), it decrementes the delay count (stopwatch), else the delaycount (stopwatch) is reset. When the delaycount reaches 0, it is assumed a valid comand has been received (a full block has been received), and the command interpreter is called. The command interpreter checks for integrity of command: it checks the block has:

- a valid command ID
- a legal length for command ID

It does not count the number of words received and compare this with the length stored as the second word. The command interpreter is written in C and the rest of the SSI code in assembler.

On a hardware error the code will:

- Reset fill pointer.
- Send out bad block.

3.2 The Blue Detector Interface

The Blue Detector Interface is a uni-directional, three wire serial interface consisting of a clock signal, envelope signal, and data signal, all of which are supplied by the Blue detector. The data from the Blue detector arrives event-by-event, with the data rate dependent on the number of photons received by the detector. The data received arrives at semi-random time intervals (Fourier analysis would reveal a power peak at the CCD frame time of 10 ms).

Blue detector data words are 24 bits, whose content is specified in XMM-OM user's manual (XMM-OM/MSSL/ML/0005.1). The parity bit is read by the DPU hardware (arbiter card). If parity is even (an even number of ones), the data word is rejected, if parity is odd, the data word is forwarded to one of the two Blue DSPs. Events are alternately sent to Blue DSP 1 and Blue DSP 2 in order for the DPU to handle the maximum data rate (i.e., event 1 is sent to Blue DSP 1, event 2 to Blue DSP 2, event 3 to Blue DSP 1, event 4 to Blue DSP 2, etc.). The arbiter card hardware selects to which of the two Blue DSPs to transmit the data. It can be commanded via the software to send all events to one or the other of the Blue DSPs, or to send the events alternately to both Blue DSPs.

3.3 The DPU – ICU Time Interface

There is a uni-directional time interface between the ICU and DPU. The ICU will supply the DPU with a time clock line and a time zero line, as defined in XMM-OM/MSSL/SP/0007, "Electronical Interfaces Specification".

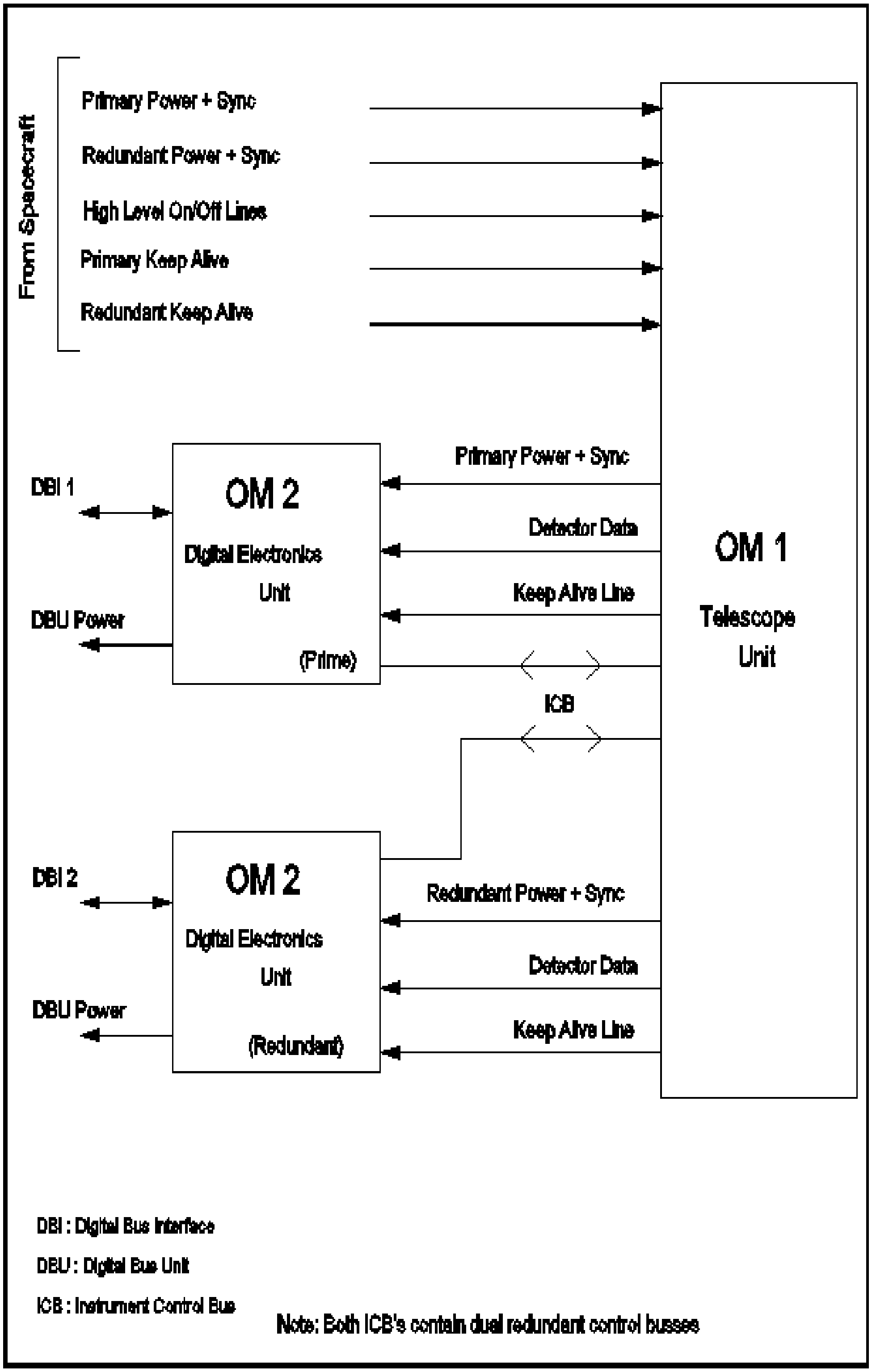

Figure 4: Context of the two DEMs (here referred to as Digital Electronic Units) and configuration of the major electrical interfaces (Figure 2.5.a of EID-B)

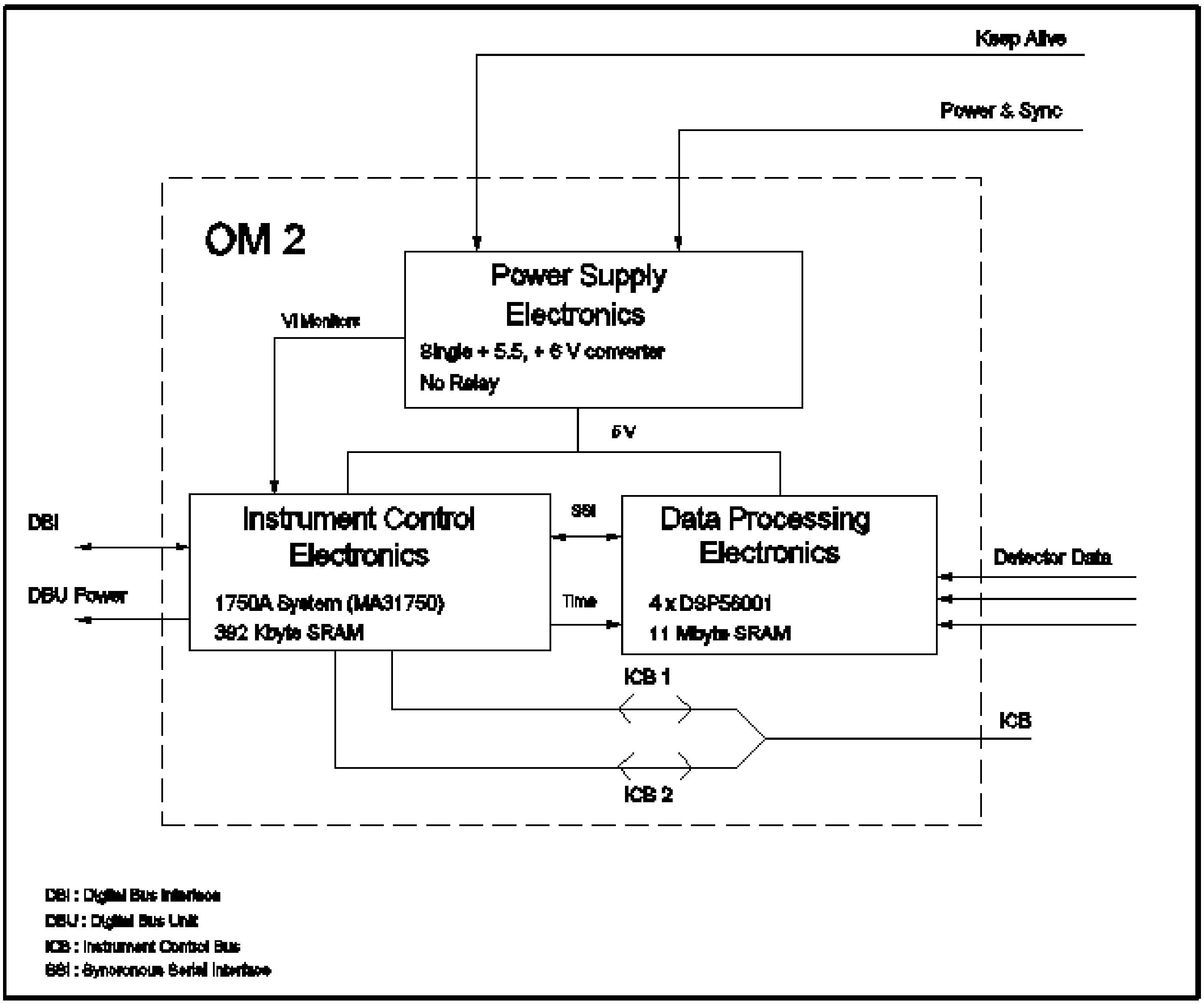

Figure 5: Context and electrical interfaces of the DPU (here referred to as the Digital Processing Electronics) within the DEM (Figure 2.5.c of EID-B)

4 System Design

4.1 Design Method

The DPU software is written in assembly language and C. The "science level" tasks are written in C to simplify software development and maintenance. Time-critical functions and the underlying operating system supporting the C environment are written in assembly language. The Blue DSP architecture is described below and further description of white DSP in addition to the Section 2.2.1 will be given in a future document, the "Detailed Architectural Design of DPU"

The DPU software can be compiled in a "simulation mode" which runs on a UNIX workstation. For details on running simulations, see the document, "XMM Optical Monitor DPU Simulation User's Manual" (XMM-OM/PENN/ML/0002). This simulation software has been developed for OM team internal use and it not part of the flight software delivered as an item. This documentation refers to the flight code implemented on the DPU flight hardware.

4.2 Blue DSP Architecture

The Blue DSPs accept and process raw data words output from the camera head Blue processing electronics. The three main processes which take place in the Blue DSPs are processing Image mode data, Raw data and Fast mode data. Image and Fast mode processing may take place simultaneously, but the input and output data from each mode are kept separate.

As part of the processing for these three modes, there are interrupt servicing routines also running on the Blue DSP's. These are the Raw data Interrupt Service Routine, Serial Communications Interface Interrupt Service routine and the Bus I/O Interrupt Service Routine. Upon interrupt, the Raw data Interrupt Service Routine places event data in the appropriate input buffer. The SCI routine handles routine communications and the Bus I/O Interrupt Service Routine handles writing processed event data over the Global I/O bus. Figure 6 illustrated the Blue DSP modes, Interface Interrupt Service routine, and background tasks.

The data flow for image mode is shown in Figure 7. For each photon, the Blue DSP must calculate a *Small Word Memory* address corresponding the where the photon arrived in the current frame, and then increment that register. Input event words are stored in an input event buffer by an interrupt driven data handler. A continuous background process, the address calculator, checks the input buffer continuously for new data. When there is data in the input buffer, each address is sequentially calculated and these addresses written to an output address buffer. A second routine checks for events in the output address buffer and increments this location of the frame in *Small Word Memory* (Frame is either in Ping or Pong Memory), when it has access to the Global I/O bus. This second routine reads the address during a bus cycle, increments the value by one and then writes the new value to this address during the next available bus cycle.

Fast mode is also processed in Blue DSP and a data flow diagram of a Fast mode is shown in Figure 8. Fast mode also uses a segment of *Small Word Memory*. Pointers to successive Fast mode frames called slices are stored in the Fast mode frame buffer. A continously resident process analyzes these slices, totalling counts received in a given slice, expressing them in compressed, differential format. These totals are written to the Fast mode data buffer. A second process writes data from the Fast mode data buffer to the *Small Word Memory* allocated for Fast Mode data when it has Global I/O bus access.

Raw mode is also processed in Blue DSP and a data flow is shown in Figure 9. Raw mode is similar to image mode in that photon events are directed to a raw data buffer by the interrupt driven event handler. A continuously running process checks the raw data buffer for events, and if they appear, divides the 24 bit event into two 16 bit words for *Small Word Memory* that are written to the raw data output buffer (The top 8 MSB are padded with 8 zeros for one word, the second word is the next 16 bits verbatim.) A second process reads these data from the raw data output buffer and writes them to *Small Word Memory* when it has access to the Global I/O bus.

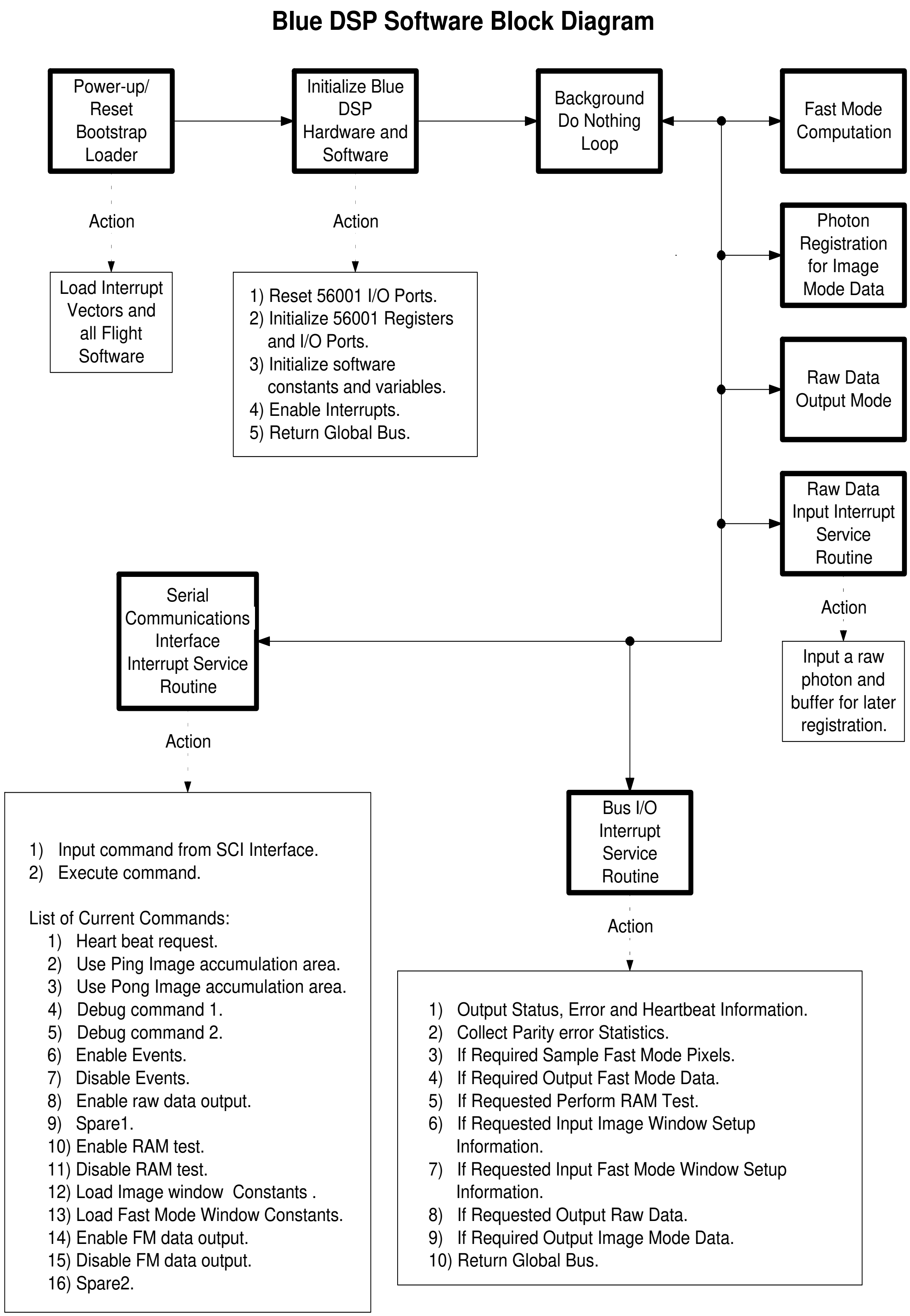
Blue DSP Software Block Diagram
Power-up/ Reset Bootstrap Loader
Initialize Blue DSP Hardware and Software
Background Do Nothing Loop
Fast Mode Computation
Action
Action
Load Interrupt Vectors and all Flight Software
1) Reset 56001 I/O Ports.
2) Initialize 56001 Registers and I/O Ports.
3) Initialize software constants and variables.
4) Enable Interrupts.
5) Return Global Bus.
Photon Registration for Image Mode Data
Raw Data Output Mode
Raw Data Input Interrupt Service Routine
Action
Input a raw photon and buffer for later registration.
Serial Communications Interface Interrupt Service Routine
Action
1) Input command from SCI Interface.
2) Execute command.
List of Current Commands:
1) Heart beat request.
2) Use Ping Image accumulation area.
3) Use Pong Image accumulation area.
4) Debug command 1.
5) Debug command 2.
6) Enable Events.
7) Disable Events.
8) Enable raw data output.
9) Spare1.
10) Enable RAM test.
11) Disable RAM test.
12) Load Image window Constants .
13) Load Fast Mode Window Constants.
14) Enable FM data output.
15) Disable FM data output.
16) Spare2.
Bus I/O Interrupt Service Routine
Action
1) Output Status, Error and Heartbeat Information.
2) Collect Parity error Statistics.
3) If Required Sample Fast Mode Pixels.
4) If Required Output Fast Mode Data.
5) If Requested Perform RAM Test.
6) If Requested Input Image Window Setup Information.
7) If Requested Input Fast Mode Window Setup Information.
8) If Requested Output Raw Data.
9) If Required Output Image Mode Data.
10) Return Global Bus.

Figure 6:

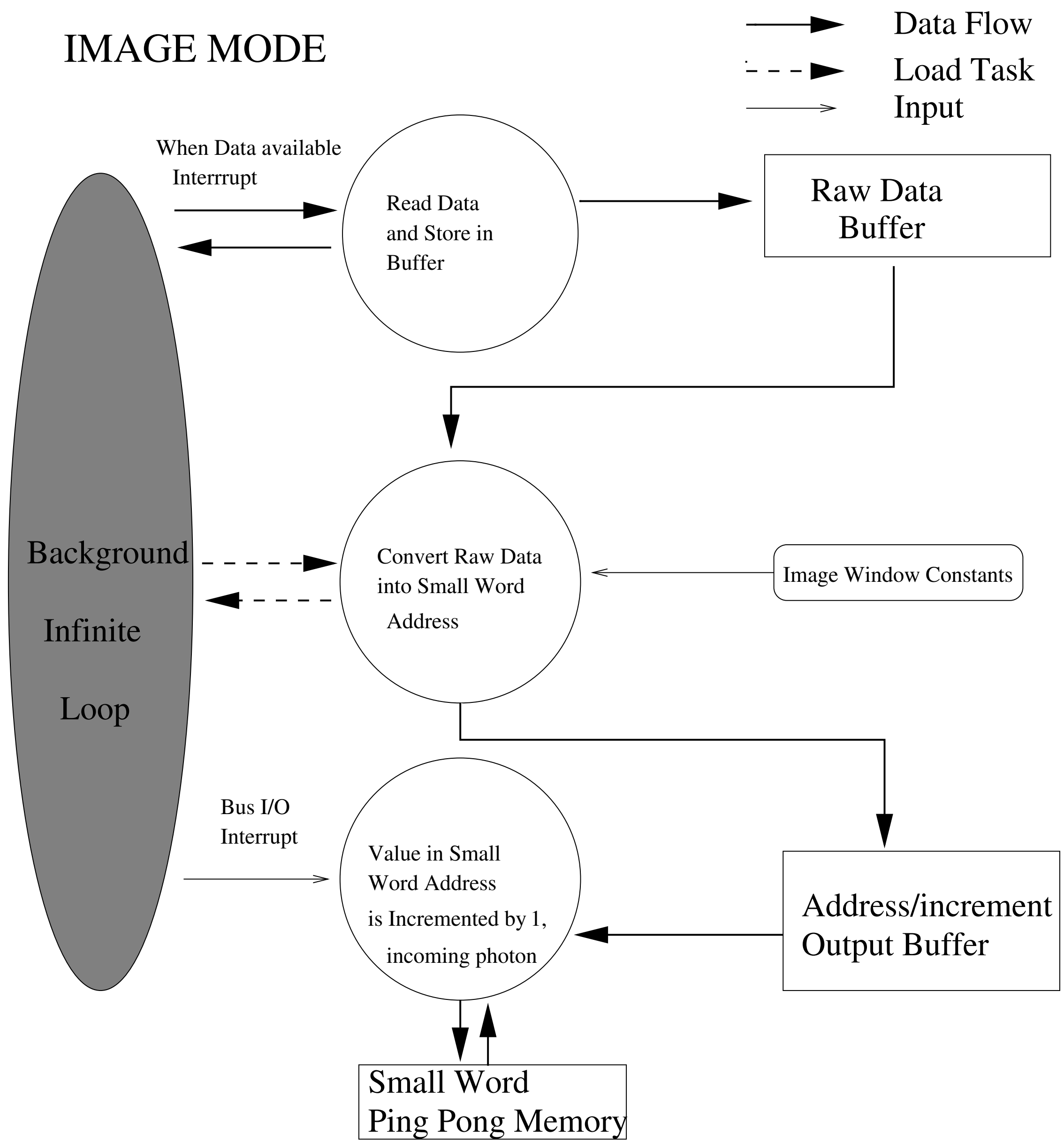

Figure 7: Image Mode diagram

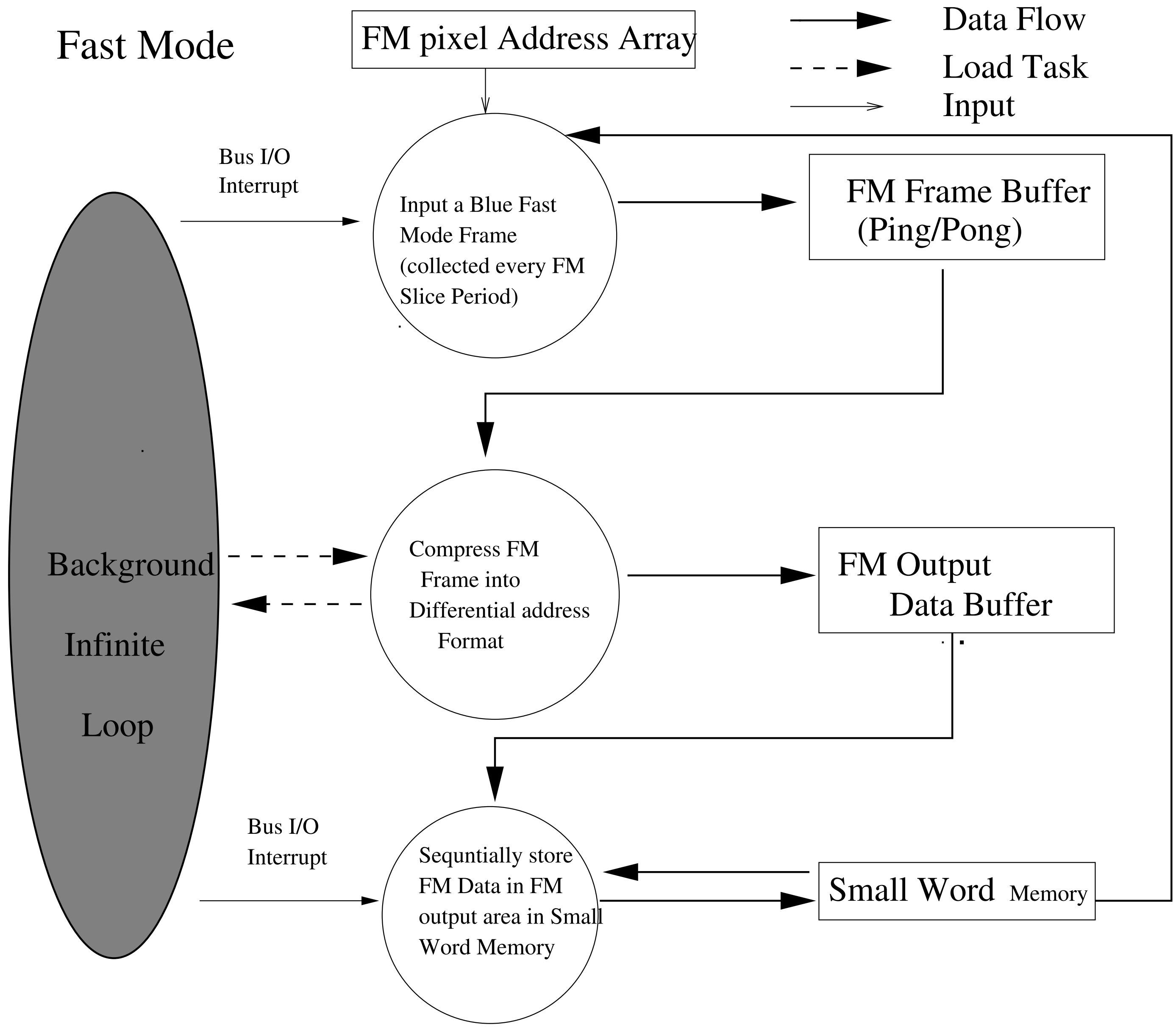

Figure 8: Fast Mode diagram

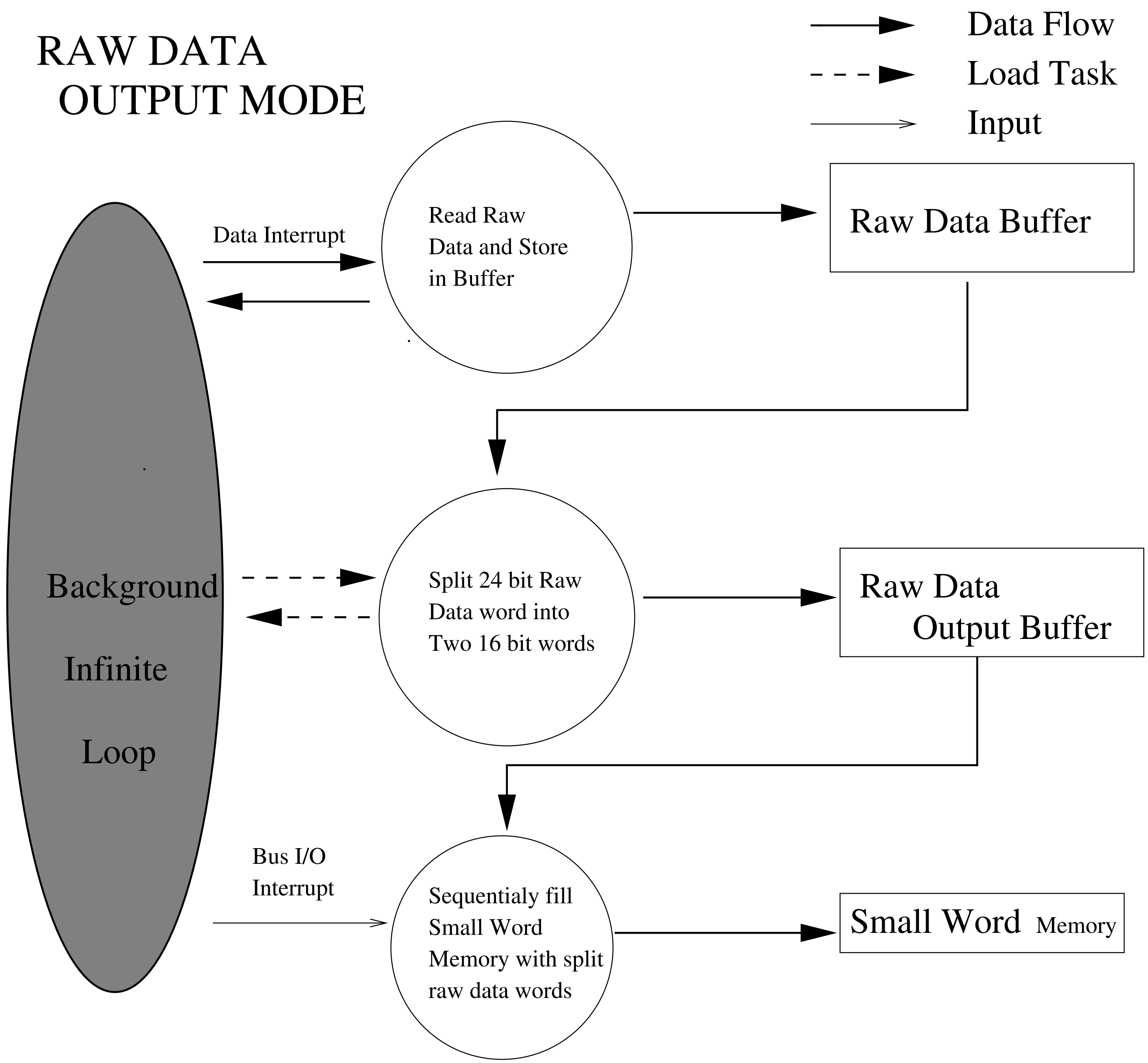

Figure 9: Raw Data Output diagram

5 Functional Description of Individual Tasks and Task Control

5.1 White DSP Software

The White DSP software is partitioned into two packages: the operating system known as DPUOS; and the White science software. The symbolic layout of the White DSP software is shown in Figure 10.

5.1.1 DPUOS:`white.asm`

5.1.1.1 Type

Assembly routine running on White DSP (part of DPUOS).

5.1.1.2 Definitions

Definitions for the assembly routines.

- *On Board RAM:* 32K X 24Bit RAM on each of the four (4) processor boards (XMM-11F).
- *On Chip RAM:* 256 X 24Bit zero wait state RAM built in the DSP56001 CPU.
- *Global Program Storage RAM:* Keep A Live (KAL) backed up 512K X 24Bit RAM on the Global Bus which provides non-volatile program storage for all four processor boards internal to the DPU.
- *SCI:* Serial Communications Interface, a serial interface protocol that is supported by circuits contained in the DSP56001 processor from Motorola. The SCI link is an "Open Drain" interface, ie. Several DSP56001s can have their respective SCI pins wired together to form a low data rate network.

5.1.1.3 Purpose and Function

Tasks which are perfomed by white.asm are listed:

1) Bootstrap Loader: Upon removal of the Reset signal (generated on the Arbiter board) the white processor executes the Bootstrap Loader to transfer executables from the "White" portion of the Global Program Storage RAM into the program execution area of the on board memory.

2) Fboot_flight

3) FecTab.asm Routine that initializes the mask patterns that are used to generate/check the Forward Error Correction Word used to verify a block of op-codes.

4)FecCode.asm

a) FfwrdErrCrection

5)xmmom_std.asm

6) BUSIO

7) Fload a) C-callable, blocks

8) Floader: a) Called from BUSIO to load and FEC a Swap Unit.

9) SSIBufferFillRoutine a) Called from BUSIO to fill a SSI output buffer b) Fills a buffer with type alert, priority, or data

10) essitx a) SSI Transmitter interrupt w/underflow Error

11) ssitx a) SSI Transmitter interrupt

12) ssirx a) SSI Recevier interrupt

13) Essirx a) SSI Recevier interrupt w/Error

14) TimeSync a) Synchronize BussCycle Count with S/C Time

15) ESCIRec a) SCI Recevier interrupt w/error

16) SCIRec a) SCI receiver interrupt

17) SCIXmit a) SCI Transmitter interrupt

18) Fsci_command a) C-callable i) Format sci command and store in buffer for output

19) Fread_n_zero_sw_int_array a) C-callable i) Setup read of small word area for next bus cycle

20) Fwrite_command_int_array a) C-callable b) Assumes caller has global bus

21) read_global

22) write_global

23) zero_global

24) SCIHbeatOutput a) ASSUMES called while white has global bus b) reads the current heartbeat response in global ram clears response in global ram checks for correct response if fail send ssi alert else point to next enabled dpu send sci heartbeat request.

25) Stkerr a) Stack error interrupt

26) Trace a) Trace interrupt

27) Illi a) Illegal interrupt

28) Swi a) Software interrupt

29) Fi__start a) Initialization on powerup

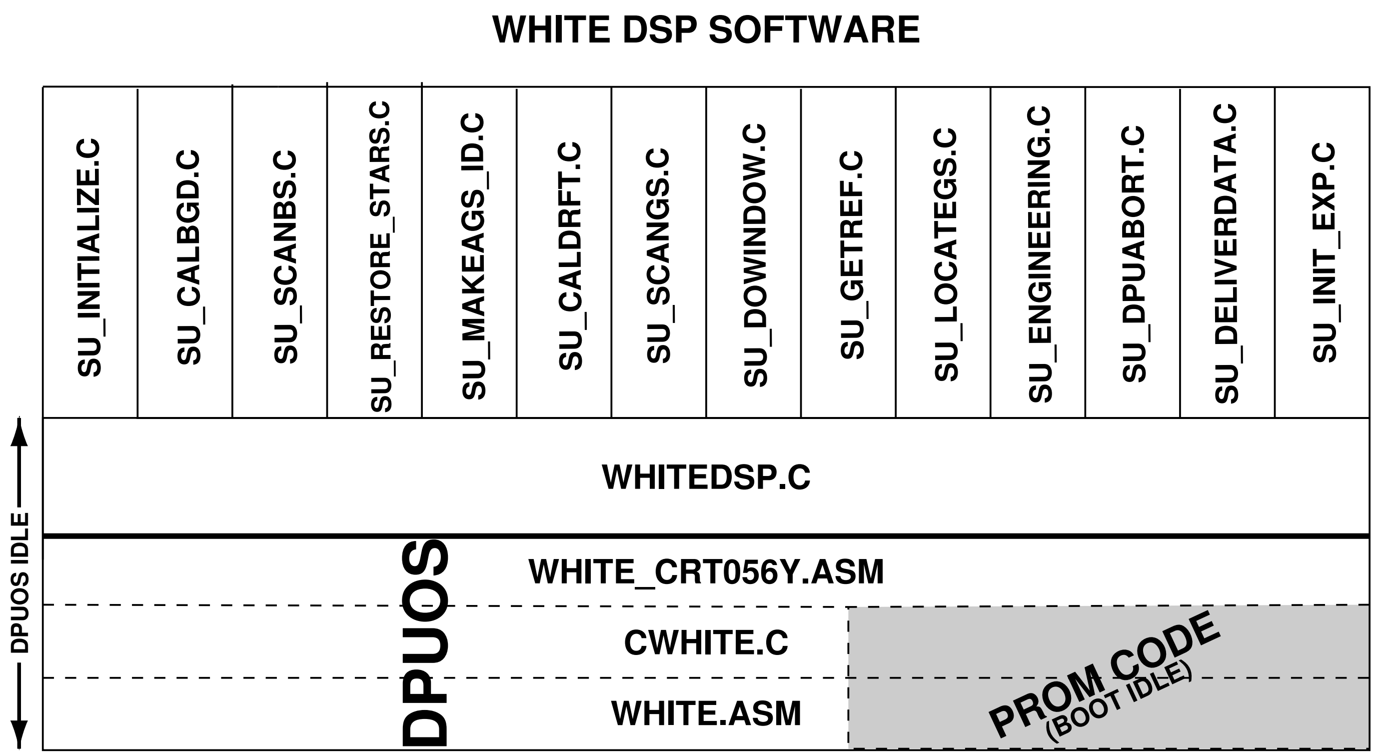

Figure 10: Symbolic layout of the White DSP software. See text for details.

5.1.2 DPUOS:`white_crt056y.asm`

5.1.2.1 Type

Assembly routine running on White DSP (part of DPUOS).

5.1.2.2 Purpose and Function

A file that is used to describe the location of all global variables used by both the C and assembly code. It also defines the location of the stack, the size of usable memory, and a few other variables used by C memory management tools.

5.1.3 DPUOS:`cwhite.c`

5.1.3.1 Type

C routine running on White DSP (part of DPUOS).

5.1.3.2 Purpose and Function

This file contains basic White DSP functions that are easiest implemented in C. The `main()` routine simply calls `whitedsp()`, which is part of the White science software (see below). An ICU command interpreter routine is part of this module.

Three alert commands are sent in this code, "Jim From The Rim", "Fred from the Dead", The supported tasks in this *swap unit* are INITIALIZE, INIT_EXP, ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, TRACK_GUIDE_STARS, COMPRESS_DATA, ENGINEERING, FINISH_FRAME. FLUSH_COMPRESS, SWITCH_PING_PONG, HDR_MANAGER, GLOBAL_ZERO, and DELAYED_Q_ALERT. Here a brief description of four tasks are listed below and the others are found in *whitedsp.c* section.

SWITCH_PING_PONG -

This function is for switching the current Blue frame memory from ping to pong or from pong to ping. BLUE DSPs send events to this frame (ping or pong) This function is used for tasks of ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, and TRACK_GUIDE_STARS. Blue frame swap the address to switch the *Small Word Memory* between ping and pong, depending on the values of the current frame.

HDR_MANAGER -

HDR_MANAGER is used to add to *PROC Memory* the DPU header such as address, ysize, yskip (bb), xsize, xskip (aa) and so on.

GLOBAL_ZERO -

This function set the PROC area to be zero.

Q_ALERT -

This function is to give out "alert message" when there is an error while execution of tasks, and the message includes the type of error, length and the format possible with string format. Since the flight code could not print out the error message directly, this Q_ALERT function is used to symbolize the errors to help tracing problems. The error message is summarized in PROTOCOL document (MSSL). This function is used particularly for tasks of WHITEDSP, CHOOSE_GUIDE_STARS and ENGINEERING.

DELAYED_Q_ALERT - .

This function calls a subroutine Q_ALERT in cwhite.c and used for whitedsp, CHOOSE_GUIDE_STARS and ENGINEERING. It gives alert message, after making a loop 12000 times.

5.1.3.3 Subordinates, Inputs and Outputs

The `cwhite.c` code includes `setup.h` and `cwhite.h`. Its `main()` routine simply calls `whitedsp()`.

5.1.3.4 Dependencies

Although the `main()` routine simply calls `whitedsp()`, other functions in this module are called from various places in the White science software and from assembly portions of DPUOS.

5.1.3.5 Interfaces

This unit includes setup.h and cwhite.h.

5.2 White Science Software

As described earlier, at power-on, the White DSP is in the DPU Ready state. Upon command by the ICU, the DPUOS is loaded with full functionality of command interpreter and I/O routines. At the end of the DPUOS loading, the base White science codes are loaded and the processing control is handed over. All other DPUOS communication operations are interrupt-driven.

The White science codes are organized in a hierarchy. At the foundation is the code `whitedsp()` performing the traffic control. It manages a number of *tasks*, which are groupings of series of processing functions. The file `whitedsp.c` contains the function `whitedsp()` and the codes for each *task*. As stated, the codes contained in the file `whitedsp.c`, including `whitedsp()` and the *task* codes, are loaded after DPUOS load and remain resident through normal science operation.

Since the *on-board memory* is limited, the processing functions in the *tasks*, which are extensive computational codes, are further grouped into discrete software modules called *swap units*. In contrast to the `whitedsp()` and the *task* codes, individual *swap units* are loaded/executed (serviced by a custom loader in the DPUOS) sequentially following the "script" in the *task* codes. Once a *swap unit* is completed, its presence in the White DSP *on-board memory* is lost, overwritten by the next *swap unit*.

A typical sequence of operation is as follows. The ICU, based on a preloaded exposure command sequence from the ground, issues a command to the DPU. The DPUOS command processor, in response to the command, changes one of the system parameters to communicate with the high-level science codes. In some cases, the system parameter is static which affects subsequent exposures only when these parameters are used. In a special class of commands for dynamic control, the "task ID" parameter is changed by the command processor. The `whitedsp()` code, upon detecting the new task ID, initiates the *task* which loads/executes the predetermined sequence of *swap units*. Upon return from the *task*, the `whitedsp()` enters the idle state/task ready for the next *task* or initiates the next *task* automatically.

It should be especially noted that the timing of major operations of the data acquisition and processing, such as exposure start/stop and *tracking frame* ping/pong swapping, is controlled by the DPU. Under normal operations, the ICU will turn on the MIC detector, or reconfigure the *detector window* configuration, and leave the MIC detector on indefinitely. A fixed length exposure is controlled by enabling/disabling the blue DSP for data acquisition event processing. The enabling/disabling of blue event processing is in turn controlled by the White DSP through the SCI interface. The codes that control the issuance of these on/off commands are residing in individual *task* codes. For example, a 10 second exposure is controlled by the *task* codes through the following sequence: send an on command to the Blue DSP(s), wait 10 seconds (a loop checking the advances of clock), then send an off command. Similarly, the switching of the *ping/pong memory* in the normal tracking sequence is also done by the *task* codes.

A limited number of "shared variables" are retained in the DPU, (*i.e.,* not swapped in and out of *on-board memory*) because they are frequently accessed by more than one swap unit and/or the OS. We distinguish between static variables, which change rarely if at all, and shared variables which change often as processing occurs. The static parameters are software configuration constants. The information transfer between *swap units* and/or the OS is through either the shared variables resident in *local memory* or the *global memory*.

5.2.1 `whitedsp.c`

5.2.1.1 Type

C routine running on White DSP.

5.2.1.2 Purpose and Function

The file `whitedsp.c` contains three parts: `whitedsp()`, task routines, and auxiliary routines (plus code used only in simulation).

As stated, `whitedsp()` handles the traffic control. It is simply an infinite loop that continuously checks the value of the dynamic variable `task_id`. Depending on `task_id`, the `whitedsp()` code executes the appropriate *task* routine. The currently supported *tasks* and brief descriptions are listed below. For more details, see the descriptions of the *swap units* invoked by the *tasks*.

IDLE -

This *task* is actually a "state" (see section 2.6). It does nothing but wait for the `task_id` variable to change. When `task_id` does change (by command from the ICU), the indicated *task* routine is invoked. Several of the *tasks* reset `task_id` to `IDLE` when they have finished, returning control to this state. There is no other code associated with this state.

ABORT -

This *task* is a place holder for emergency exit. Detailed operations are TBD. Currently its actions are to issue a `DA_DPU_ABORTING` alert, to disable Blue image and fast mode event collection, and to set the `task_id` to `IDLE`. It is invoked in a few ways: by an `IC_ABORT_DPU` command from the ICU; in the `whitedsp()` loop if an invalid `task_id` is seen; and by the TRACK_GUIDE_STARS and FINISH_FRAME *tasks* if `task_id` is not `TRACK_GUIDE_STARS` or `FINISH_FRAME`, respectively, at the end of their operations. This *task* calls one *swap unit*: DPUABORT.

INITIALIZE -

The purpose is to initialize all parameters to default values. Note that INITIALIZE is not a formal *task*. It is distinguishable from the other *tasks* in that it normally will be run only at DPU boot-up, or if the White DSP is reset. It is invoked by an `IC_INIT_DPU` command from the ICU, and sends a `DA_EOT_INIT_DPU` alert when finished. This *task* calls one *swap unit*: INITIALIZE.

INIT_EXP -

This *task* is run immediately before field acquisition. Its purpose is to prepare the DPU to obtain the *acquisition frame* by setting up the appropriate *detector window* configuration, which is also used when obtaining the *reference frame* (see the CHOOSE_GUIDE_STARS *task* below). It is invoked by an `IC_INIT_EXP` command from the ICU, and sends a `DA_EOT_INIT_EXP` alert when finished. This *task* calls one *swap unit*: INIT_EXP.

ACQUIRE_FIELD -

This *task* determines the relative offset between the commanded satellite pointing and the actual pointing. First, a short exposure (nominally 10 seconds), called the *acquisition frame* is obtained, typically with the filter wheel in the V filter position. That image is scanned for bright sources. A double iterative procedure is performed to 1) identify the observed bright stars with the previously up-linked *absolute guide stars*, and 2) calculate the relative offset. A robust criterion is used to determined the convergence of solution. The final product consists of three numbers giving the translation offset (*acquisition offset*) and roll angle between actual and commanded pointing. If field acquisition fails, the *AO* values are set to zero.

This *task* is invoked by the `IC_ACQUIRE_FLD` command from the ICU. After the *acquisition frame* has been obtained, it sends a `DA_DONE_FLD_DATA` alert. When finished with its field acquisition calculations it sends a `DA_EOT_ACQUIRE_FLD` alert. It calls the following *swap units*: CALBGD, CALDRFT,

DELIVERDATA, MAKEAGS_ID, RESTORE_STARS, and SCANBS.

CHOOSE_GUIDE_STARS -

This *task* serves three major purposes. It: 1) controls acquisition of the *reference frame*; 2) selects *guide stars* for tracking during the *exposure* which follows; and 3) sets up the the appropriate window and memory configuration. A nominal $\sim$ 10 second *frame* is taken (again controlled by the task code itself) with the observer-selected filter. As in the ACQUIRE_FIELD *task*, bright sources in the reference frame are scanned and analyzed. The analysis gives a number of "quality" indicators for the bright sources. Based on these quality indicators and other global considerations such as star crowdedness, a number of *guide stars*, nominally 10, are selected. These star positions are saved for subsequent tracking. Based on the user-supplied/requested window configuration, the selected *guide stars* and the *acquisition offset*, a window/memory management code configures *detector windows* to coordinate data acquisition, *memory windows* to store acquired raw data, and *science windows* to specify on-board scientific data processing. The user-requested *science window* coordinates are shifted to account for the *acquisition offset*. The *memory windows* are also shifted, but by an amount as close to the *AO* as possible while obeying particular *memory window* restrictions. Those restrictions are: 1) that *memory windows*' BLC coordinates be multiples of 16, and 2) that *memory window* sizes also be multiples of 16. These restrictions are imposed due to the requirements of event centroiding.

In the event of *guide star* selection failure, i.e., too few stars meeting the quality criteria, the code currently takes no special action. It has been proposed to proceed with the *exposure* with the tracking calculations and subsequent shift in the *shift-and-add* process disabled. In this case, effective loss of spatial resolution may be apparent. This has not yet been implemented.

If no user-defined windows are validated, there is no point in proceeding with an exposure. In this case, it has been proposed that the DPU attempt to set up a default window configuration and proceed with the *exposure*. This has not yet been implemented.

This *task* is invoked by the `IC_CHOOSE_GS` command from the ICU. When finished, it sends a `DA_EOT_CHOOSE_GS` alert. It calls the following *swap units*: CALBGD, DOWINDOW, DELIVERDATA, GETREF, RESTORE_STARS, SCANBS, and SCANGS.

TRACK_GUIDE_STARS -

This *task* is invoked by an `IC_TRACK_GS` command from the ICU at the beginning of an *exposure* (at which point it sends a `DA_BEGOF_EXP` alert), and subsequently by the FINISH_FRAME *task* if the *current exposure* is not complete. It calculates the drift offset and roll of a just-completed *tracking frame* relative to the *reference frame*. This is done by first calculating the position of the *guide stars* in the *tracking frame* in question. Then a maximum likelihood solution is found by comparing the *guide stars'* calculated and reference position. Once the translation offset has been calculated, this *task* commands the Red DSP into its RED_ACCUMULATE_IMAGE *task* to perform the *shift-and-add* operation for the image mode *science windows*. Then `task_id` is set to `COMPRESS_DATA` and this *task* exits. After completion of an *exposure*, but before calculation of the tracking solution for the final *tracking frame*, it sends an `DA_ENDOF_EXP` alert. This allows the ICU to terminate detector integration and to move the filter wheel if necessary. This *task* calls the following *swap units*: CALBGD, CALDRFT, DELIVERDATA, LOCATEGS, and RESTORE_STARS.

COMPRESS_DATA -

This *task* is not executed in response to a command from the ICU. Rather, it is automatically invoked at the end of TRACK_GUIDE_STARS. It invokes the COMPRESS *swap unit* to compress previous exposure data which have been placed in the compression queue. Only the `DD` data types are compressed. Compression is done a block at a time in a loop so that this *task* can keep track of the progress of the *current tracking frame*. When the current *tracking frame time* is nearly complete (within `COMPRESS_TIME_BUFFER` milliseconds of the end of the *current tracking frame*), control passes out of the loop so the status of the *shift-and-add* operation can be monitored. As soon as *shift-and-add* is done the `task_id` is set to `FINISH_FRAME` and this *task* exits, allowing FINISH_FRAME to run. This procedure ensures that a new *tracking frame* will not be started until data for the *previous tracking frame* have been co-added into the *accumulating image*. This *task* calls the following *swap units*: COMPRESS.

FINISH_FRAME -

This *task* is not executed in response to a command from the ICU. Rather, it is automatically invoked at the end of COMPRESS_DATA. This *task* waits until the Red DSP has finished the *shift-and-add* calculation, at which time memory for another *tracking frame* becomes available. It then then checks to see whether the *exposure* is complete. If it is, the *task* calls either the DELIVERDATA or the SUBMIT_CMPRSS_Q *swap unit* to place housekeeping data and the *accumulated image* data on the data queue, sends a `DA_COMPLETE_EXP` alert, sets `task_id` to IDLE, and exits. If the *exposure* is not complete, the *task* waits until the current *tracking frame time* has expired, sets `task_id` to `TRACK_GUIDE_STARS` and exits so that the next *tracking frame* will be obtained. This *task* calls the following *swap units*: DELIVERDATA or SUBMIT_CMPRSS_Q.

FLUSH_COMPRESS -

This *task* is executed in response to an `IC_FLUSH_CMPRS` command from the ICU, and sends a `DA_EOT_FLUSH_CMPRS` alert when finished. Because compression of exposure n of a pointing will occur during *shift-and-add* operations for exposure $n+1$, this task will normally be invoked only after the last exposure of a pointing. [What about engineering mode data?] This *task* calls the following *swap units*: COMPRESS.

ENGINEERING -

This *task* sets up the window configuration required for the particular engineering mode function requested, and issues `ENABLE_EVENTS` and `DISABLE_EVENTS` to the Blue DSPs to control the effective exposure time. There are seven engineering modes. Modes 0, 1, and 2 are for delivery of raw MIC data (no memory address calculation and increment) for both, Blue 1, and Blue 2 DSPs, respectively. Mode 3 is for obtaining M,N data for calculation of a new centroiding look-up table (LUT). Mode 4 is for obtaining a full frame image at high resolution. Mode 5 also acquires a full frame at high resolution, but processes it to determine whether the current LUT is good. Mode 6 is for obtaining data on intensifier characteristics. This *task* is executed in response to an `IC_ENBL_ENG` command from the ICU, and sends a `DA_EOT_ENG` alert when finished. This *task* calls the following *swap units*: DELIVERDATA, and ENGINEERING.

Each of the tasks described above, with the exception of IDLE, is contained in a separate routine in the `whitedsp.c` file. In addition to invoking individual *swap units* in sequence for data processing, high-level control codes such as timing, SCI commanding of the Blue DSP, data I/O queueing to the ICU, and sending time-critical high-priority alerts to the ICU are all contained in the *task* codes.

In addition to the alerts specified in the *task* description above, each *task* optionally (if verbose logging is enabled) sends "end of *swap unit*" alerts (`DA_EOS_*` where `*` is a wildcard for the *swap unit* names).

In addition to `whitedsp()` and the *task* routines, the file `whitedsp.c` also includes several auxiliary codes that are used by a number of *task* codes and/or *swap units*. These include a manager for headers for production data, a routine to zero out large area of memory, a routine to switch ping pong *tracking frame*. And a suite of memory access wrapper codes described below.

The `whitedsp.c` code "includes" a file `global_access.c` containing a full suite of C wrapper codes for memory access. As a result of the system design for the global memory access shared among four processors, the read/write access of the global memory needs to be done explicitly in the science codes, with careful timing considerations. There is also a limit to the number of read/write operations that the code can perform at a time such that a single processor does not tie up the available global memory access. (The case of a processor tying up the memory access for whatever reason is considered a fault condition trapped by the DPUOS and the DPU will undergo a hardware reset.) To facilitate memory access and code management, the global memory access are provided in a suite of C-callable codes. These codes follow the naming convention of [`read/write`]_[`sw/bw/proc`]_[`int/float`](`_array`) and are contained in the file `global_access.c`. The *global memory* access codes are in fact C wrapper codes which check for the validity of requested range for memory prior to invoking a low-level code engine written in assembly for the memory access.

5.2.1.3 Subordinates

The `whitedsp.c` code includes files `setup.h` and `global_access.c`. When compiled for simulation on a Sun platform, it also includes `shared_variables.c`, `sun_util.c`, and `shared_io_functions.c`.

5.2.1.4 Dependencies

The `whitedsp.c` code is the main traffic control code for the White DSP. The main task functions are part of this source module. The high level function, `whitedsp()`, is called from the `main()` routine in `cwhite.c`.

5.2.1.5 Interfaces

The White DSP traffic control code is invoked thus: `whitedsp();`. In nominal operation it will never return to the calling routine.

5.2.2 `shared_io_functions.c`

5.2.2.1 Type

C code written for white, red, and blue DSP.

5.2.2.2 Purpose and Function

This swp unit reads and writes the White DSP local RAM file with standard comment header, shell (?) for Red local RAM. This unit has following functions.

READ_SHARED_DATA - This function is to open the local RAM file for reading.

WRITE_SHARED_DATA - This function is to open the local RAM file for writing.

GLOBAL_RAM_IO - This function is to read from the global RAM file.

PROC_RAM_IO - This function is to read from the $PROC$ RAM file. This will be used in mutual exclusion with global_ram_io.

LOCAL_RAM_IO - This function is to read the local RAM file.

BLUE_LOCAL_RAM_IO - This function is to read from the BLUE local RAM file.

RED_LOCAL_RAM_IO - This function is to read from the RED local RAM file.

5.2.2.3 Subordinates

This module includes stdio.h, sim_blue_local_ram.h, and sim_red_local_ram.h.

5.2.2.4 Interfaces Calling sequence is integer_variable, read_shared_data.

5.2.3 `shared_variables.c`

5.2.3.1 Type

C code written for White DSP.

5.2.3.2 Purpose and Function

Declaration of White DSP local RAM variables such as address of frame, type of filter on blue detector, field acquisition, shift-and-add status on Red DSP, data delivery, variables assigned to symbolic constants that are defined in white_shared_variables.h, global variables for compression. The variables that are changed by the hardware, such as real-time clocks, interrupts and DPUOS command processor are also defined. The unit writes all header information for house keeping such as observation date, and exposure time.

5.2.4 `su_calbgd.c`

5.2.4.1 Type

C code written for the White DSP *swap unit* CALBGD.

5.2.4.2 Purpose and Function

This *swap unit* calculates the background for a *tracking frame*. It samples data from three distinct regions, equally spaced in the previous *tracking frame* (stored in ping or pong), and calculates the average number of counts per pixel. Ths average is then used to set up the threshold level used in the stellar image construction function. This sampling is performed independent of the window dimensions, which is optimized for efficient use of the global bus.

Three major step executions can be described as the following: i) When the task is for ACQUIRE_FIELD or CHOOSE_GUIDE_STARS, its frame is set to full frame to determine the number of pixels in the frame, based on which number of pixels per block and the separation between blocks are determined. ii) After initialize address, counters and array access parameters in previous frame, accumulate counts as a sum. iii) The threshold level of star recognition is set to be the number of pixel block multiplied by the average number of counts per block.

5.2.4.3 Subordinates

This module includes `setup.h`.

5.2.4.4 Dependencies

The *swap unit* CALBGD is used in the *tasks* ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, and TRACK_GUIDE_STARS.

5.2.4.5 Interfaces

The CALBGD *swap unit* is invoked thus: `return_status = execute (SWAP_CALBGD);` where `#define execute load` is in effect in the flight version.

5.2.4.6 Resources

This *swap unit* requires (`N_PIX_BLKS + 1`) DPU cycles, plus an additional cycle if a log is kept. It requires approximately 300 words of DSP RAM.

5.2.5 `su_caldrft.c`

5.2.5.1 Type

C code written for the White DSP *swap unit* CALDRFT.

5.2.5.2 Purpose and Function

This *swap unit* calculates the best fit values ΔX, ΔY, and $\sin\Theta$ needed to make the observed "current" positions of the *guide stars* or *absolute guide stars* agree with their nominal positions. Sporadic outliers in calculated *guide star* positions are handled/eliminated through iterative assignment of statistical weighting depending on the star's calculated position and calculated drift (see XMM-OM/PENN/TC/0004).

The *current coordinates* are assumed to be related to the *nominal coordinates* by a rotation/translation transformation as if, relative to the detector frame of reference at the time of observation, the detector needs to undergo a translation by ΔX pixels in x and ΔY pixels in y, then a rotation by angle Θ (assumed to be small) about a roll center at (X_0, Y_0) in the shifted coordinates, for consistency with the *nominal coordinates*. Thus,

$$[(X_{current} - X_0) \;=\; (X_{nominal} - X_0)\cos\Theta \;-\; (Y_{nominal} - Y_0)\sin\Theta \;+\; \Delta X]$$

and

$$[(Y_{current} - Y_0) \; = \; (X_{nominal} - X_0)\sin\Theta \; + \; (Y_{nominal} - Y_0)\cos\Theta \; + \; \Delta Y]$$

Note that X_0 and Y_0 are not fitted parameters; they are constants which give the location of the rotation axis in the detector field of view.

The best fit parameter values are the values that minimize the sum of the squared differences between the observed star positions and the "current" positions calculated using these transformation equations.

In the ACQUIRE_FIELD *task*, a subset of the *absolute guide stars* consisting of those identified by MAKEAGS_ID are used in the calculation. The *nominal coordinates* are those of the (identified) *absolute guide stars* specified relative to the commanded satellite orientation, and the *current coordinates* are those of the corresponding stars as observed in a field acquisition *reference frame* (see `su_makeags_id.c`). Equal weights are assigned initially to the stars. The resulting parameters ΔX, ΔY, and $\sin\Theta$ give the "absolute" transformation needed to center the *science, memory,* and *detector windows* on various targets.

In the TRACK_GUIDE_STARS *task*, the *nominal coordinates* are those of a set of selected *guide stars* as observed in the initial *reference frame*, and the *current coordinates* are those of the same stars as located in a subsequent *tracking frame*. Only *guide stars* of "good quality" are used. (Note that, while SCANGS *swap unit* selects a set of *guide stars* which initially are "good," there is no guarantee that a star will remain "good" in all subsequent *tracking frames*). The weight assigned initially to each of the *guide stars* is the number of counts in the psf recorded during the *reference frame*. The resulting parameters (ΔX, ΔY) give the "shift" required to compensate for satellite drift (the roll is not used for this purpose) to accumulate an image for a time longer than that over which significant drift occurs (the so-called "*shift-and-add*" function).

ΔX and ΔY are determined assuming that the roll is small but possibly non-zero. Specifically, it is assumed that $\cos\Theta \; \simeq \; 1$, in which case

$$[X_{current} \; \simeq \; X_{nominal} \; - \; (Y_{nominal} - Y_0)\sin\Theta \; + \; \Delta X]$$

and

$$[Y_{current} \; \simeq \; Y_{nominal} \; + \; (X_{nominal} - X_0)\sin\Theta \; + \; \Delta Y]$$

In the *shift-and-add* function (`su_saa`) the roll is neglected altogether; there it is simply assumed, in effect, that

$$[X_{current} \; \simeq \; X_{nominal} \; + \; \Delta X]$$

and

$$[Y_{current} \; \simeq \; Y_{nominal} \; + \; \Delta Y]$$

5.2.5.3 Subordinates

This module includes `setup.h`.

5.2.5.4 Dependencies

The *swap unit* CALDRFT is used in the *tasks* ACQUIRE_FIELD and TRACK_GUIDE_STARS.

5.2.5.5 Interfaces

The CALDRFT *swap unit* is invoked thus: `return_status = execute (SWAP_CALDRFT);` where `#define execute load` is in effect in the flight version.

5.2.5.6 Resources

This *swap unit* requires about 1 DPU cycle, and about 70 words of DSP RAM.

5.2.5.7 References

See XMM-OM/PENN/TC/0004.03 and XMM-OM/PENN/TC/0021.01.

5.2.6 su_compress.c

5.2.6.1 Type

C code written for the White DSP *swap unit* COMPRESS.

5.2.6.2 Purpose and Function

This *swap unit* is used to compress the DD data types so that the data may be transmitted to the ground within the OM telemetry allocation. It is used in the COMPRESS_DATA *task*.

If there are entries on the compression queue, the data are compressed using the Variable Block Tiered Word Length (VBTWL) algorithm, which is described in section V of "DPU Processing for XMM/OM, Tracking and Compression Algorithm, Document 3," which is project document XMM-OM/PENN/TC/0004.03. Compressed data are then placed on the regular data queue for delivery to the ICU.

Briefly, the compression algorithm works as follows: The two basic phases are decorrelation and encoding. Decorrelation means taking out any "predictable" or "repeatable" portion of the data in an image. For OM data, this can be viewed as the mean background level. The residual image, which consists largely of Gaussian excursion from the background level (standard deviation σ) and outliers due to stellar images. Encoding with VBTWL allows us to encode the bulk of the distribution (the many background pixels) with few bits per datum, and the relatively small number of outlier pixels (pixels in star images) requiring more bits per datum. We aim to group $\pm 3\sigma$ into the lowest tier (99% of background pixels). Outliers, the remaining background and stellar pixels, will be grouped in the next two tiers.

The total volume of compressed data for an *image mode window* will be approximately $n_x n_y \log_2(k(N/n_x n_y)^{1/2})$ bits, where N is the total number of counts in the window, n_x and n_y are the x and y dimensions of the window in pixels, and k is roughly 6 in the scheme outlined above. For a typical 1024×1024 window (binned to 512×512) using a U filter, N will typically be about 4 million counts in a 1000 second exposure. More than two thirds of that will be background. This results in somewhat less than 1 million bits. At 8 kbit s^{-1}, the compressed data can be transmitted in about two minutes.

5.2.6.3 Subordinates

This module includes `setup.h`, `vbwtl.h`, and `vbtwl.def`.

5.2.6.4 Dependencies

The *swap unit* COMPRESS is used in the *tasks* COMPRESS_DATA and FLUSH_COMPRESS.

5.2.6.5 Interfaces

The COMPRESS *swap unit* is invoked thus:

```
return_status = execute (SWAP_COMPRESS);
```

where `#define execute load` is in effect in the flight version.

5.2.7 su_deliverdata.c

5.2.7.1 Type

C code written for the White DSP *swap unit* DELIVERDATA.

5.2.7.2 Purpose and Function

This *swap unit* is a collection of data delivery routines for all data types. Through the global variable `data_delivery`, the invocation of this *swap unit* will construct and format the appropriate data sets for delivery to the DPUOS data output queueing system. Note that DD data types will normally not be handled by DELIVERDATA, but will be placed on the compression queue by SUBMIT_CMPRSS_Q and ultimately placed on the data queue by the COMPRESS *swap unit*.

For priority data, the data are copied into a predetermined area from where the DPUOS pick them up and sends them to the ICU, with medium priority. There can only be one priority data set being sent at a time. If there is an existing priority data set, then the processing will wait until the previous data set is fully sent allowing the current priority data set to be inserted into the queue.

For production data, the data remain in situ with a separate header constructed by the header manager. The queue for production data is actually a pointer management system that actively alerts the ICU of the presence of production data. The ICU, upon being alerted of pending production data, will request the data from the DPU. The DPU responds by sending the requested number of data blocks. In the current implementation, the *swap unit* calls a DPUOS-provided C routine `data_queue()`. This routine simply inserts the appropriate pointers to the queueing buffer. The actual transmission is done as a low-level interrupt-driven function, coded in assembly. The data transmission assembly code will zero out the memory where the production data were located after transmission.

5.2.7.3 Subordinates

This module includes `setup.h`.

5.2.7.4 Dependencies

The *swap unit* DELIVERDATA is used in the *tasks* ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, TRACK_GUIDE_STARS, FINISH_FRAME, and ENGINEERING.

5.2.7.5 Interfaces

The DELIVERDATA *swap unit* is invoked thus:

```
return_status = execute (SWAP_DELIVERDATA);
```

where `#define execute load` is in effect in the flight version.

5.2.8 su_dowindow.c

5.2.8.1 Type

C code written for the White DSP *swap unit* DOWINDOW.

5.2.8.2 Purpose and Function

This is a *swap unit*, used only in the CHOOSE_GUIDE_STARS *task*; the purposes are (a) to define *memory, science, and detector window* parameters corresponding to user-specified *science* and *memory windows*; (b) to establish up to 10 (`MAX_TRACKING_WINDOWS`) *science windows* and, if necessary, additional *memory* and *detector windows* corresponding to a set of *guide stars* for tracking; and (c) manages *Big-word memory* for DPU processed data and telemetry storage.

First the user-specified windows are shifted to take the estimated error in absolute pointing into account (the window coordinates specified by the observer are relative to the commanded spacecraft pointing direction, i.e., the *acquisition offset*). If the *acquisition offset* shift would move a window partially off the detector area, it will be resized accordingly, i.e., the portion shifted off the detector will be trimmed. Any *memory window* trimmed to less than 32 detector pixels in x or y will be rejected.

Memory window BLC coordinates and sizes must be multiples of 16. This is required to facilitate the photon even centroiding algorithm. As a result, *memory windows* may not be shifted by exactly the amount of the *acquisition offset*. The difference may be as much as 8 pixels in x and y. *Science windows* will be shifted by the amount of the *acquisition offset*. Thus, to avoid rejection of *science windows*, their corresponding *memory windows* should provide "buffers" of at least 8 pixels. Larger buffers are recommended to allow for migration of *science windows* within *memory windows* due to spacecraft drift during the exposure.

The validity of user-specified windows is checked (*e.g., memory windows* must be non-overlapping and must fit in the memory available), *science windows* are matched to the appropriate *memory windows*, and *detector windows* are established to cover the *memory windows*. (Note: some of the validation functionality would become unnecessary if a suitable ground-based window configuration program were made available to observers.)

The quality of an otherwise good *guide star* is set to `CROSS_TKW_QUALITY` (a value recognized as "bad") if the square *memory window*, of size `TRACKING_MMW_DIM` (64 pixels), required to enclose it crosses a valid user-specified *memory window*. Any other potentially good *guide stars* to which *science, memory* and *detector windows* are not assigned (*e.g.,* because the number of good *guide stars* chosen by SCANGS exceeds `MAX_TRACKING_WINDOWS`) are given a quality value 2 × `CROSS_TKW_QUALITY`, which is also interpreted as "bad."

Windows are subject to the following additional constraints:

- The sum of memory usage of all memory windows and fast mode (processed) data should not exceed X 16-bit words (X is TBD, should be between 2 to 4 M).
- The sum of memory usage of all image mode accumulation data should not exceed Y 24-bit words (Y is about 0.5 M, TBC), and
- The maximum number of *science windows* allowed is 16, though this is somewhat arbitrarily set. The maximum number of memory windows is also set at 16. The number of *detector windows* on either detector is set by the detector interfaces at 15 for the MIC detector. In full frame engineering mode, 16 512 × 512 *detector windows* will be required. Special setup for this case is done by the ENGINEERING *swap unit*.
- *Detector windows* may not exceed 512 pixels in either dimension. Thus, an 800×800 *memory window* would require four *detector windows* of the following sizes: 512×512, 288×288, 512×288, and 288×512.
- *Memory windows* may not overlap.

5.2.8.3 Subordinates

This module includes `setup.h`.

5.2.8.4 Dependencies

The *swap unit* DOWINDOW is used in the *task* CHOOSE_GUIDE_STARS.

5.2.8.5 Interfaces

The DOWINDOW *swap unit* is invoked thus:

```
return_status = execute (SWAP_DOWINDOW);
```

where `#define execute load` is in effect in the flight version.

5.2.8.6 Resources

Global RAM access requirements for this *swap unit* are:

- Up to 258 words to read user specified window data
- Up to 258 words to write *memory* and *science window* data.
- Up to about 65 (TBD) words to write *detector window* data
- A few additiona words if a log is kept

This amounts to 2 or 3 DPU cycles.

This *swap unit* requires approximately 450 words of DSP RAM.

5.2.8.7 References

See related document on observing mode requirements.

5.2.9 `su_dpuabort.c`

5.2.9.1 Type

C code written for the White DSP *swap unit* DPUABORT.

5.2.9.2 Purpose and Function

This *swap unit* is a place holder for emergency exit handling, If this swap unit doesn't do anything, then returns to the calling routine with status = 0.

5.2.9.3 Subordinates

This module includes `setup.h`.

5.2.9.4 Dependencies

The *swap unit* DPUABORT is used in the *tasks* DPUABORT.

5.2.9.5 Interfaces

The DPUABORT *swap unit* is invoked thus:

```
return_status = execute (SWAP_DPUABORT);
```

where `#define execute load` is in effect in the flight version.

5.2.10 `su_engineering.c`

5.2.10.1 Type

C code written for the White DSP *swap unit* ENGINEERING.

5.2.10.2 Purpose and Function

This *swap unit* performs the window setup for the five engineering modes, and writes them into PROC memory so that engineering data are written into known memory locations. It is used only in the ENGINEERING *task*. For the centroiding LUT confirmation mode it performs necessary processing of a full-frame high resolution frame to produce the required 8×8 arrays for each of 16 512×512 sectors of the detector (see section 2.7.4). In the engineering modes, the ICU dictates the portion of the detector to use; the output engineering data will be compatible with the window parameters defined here.

Window parmaters are determined for the address which equation is given in section 3.7.3 of XMM-OM User Manual Part 1A - Experiment On-Board Software - ICU (XMM-OM/MSSL/SP/0005.1). The parameters of window setup, number of window, offset from the window (aa), and size of window (bb) determine the address of each mode.

Mode 0, 1 and 2 (ENG_RAW_DATA_B1 and ENG_RAW_DATA_B2): from the event MIC file, the output has the same information as the input but with a format of positions of x, and y, and number of photons.

Mode 3 (CENTROID_LUT_CALC) calculates a set of channel boundaries (9 numbers) scaled by a thousand, where the channel boundary is boundaries of subpixels. The boundaries of the subpixel are corrected to get an high quality image, using a look up table of M,N images. M, N images (sub-type data 7) are defined as a function of the detected counts for a test pixel. The definition of M, N are: M=c-a and N=4b-2a-2c, where a,b,c are distribution of detected photons over the subjunct pixels. The following four major task are performed as in order: i) read the M,N image, an input file which is created from electronics, ii) estimate the histogram of the number of photons for a given value of M/N, and iii) calculate the integrated value of the photon numbers as the number M/N increases, and iv) based on the final sum of integrated value, each boundary is adjusted using 1/8 of the final sum value.

Mode 4, 5, and 6: Mode 4 (ENG_FULL_FRAME) creates a full resolution image. The output is 262144 numbers shich can be converted to 2048x128, which of 16 created by 16 windows will make a full resolution image of 2048x2048 pixel image. Mode 5 (CENTROID_LUT_CONF) is centroiding confirmation mode. The centroid look up table confirmation requires a full frame at high resolution, and the detector centroiding image corresponds to the input image. Mode 6 (ENG_PULSE_HEIGHT) creates output of a height of pulse obtained by electronics.

5.2.10.3 Subordinates

This module includes `setup.h`. When compiled for use in simulation on a Sun platform, it also includes `sim_blue_local_ram.h`. An input file of M, N image is required as an input and the image is obtained from electronics.

5.2.10.4 Dependencies

The *swap unit* ENGINEERING is used in the *task* ENGINEERING.

5.2.10.5 Interfaces

The ENGINEERING *swap unit* is invoked thus:
`return_status = execute (SWAP_ENGINEERING);`
where `#define execute load` is in effect in the flight version.

5.2.11 `su_getref.c`

5.2.11.1 Type

C code written for the White DSP *swap unit* GETREF.

5.2.11.2 Purpose and Function

This *swap unit*, used only in the CHOOSE_GUIDE_STARS *task*, establishes the roll center for tracking. There are three possibilities for roll center designation: center of field of view, "center of mass" amongst guide stars, and user-specified location. The default location is the the center of mass of guide stars. The locations of guide stars are then adjusted to be relative to the roll center. This saves a little bit of time in the CALDRFT *swap unit*.

5.2.11.3 Subordinates

This module includes `setup.h`.

5.2.11.4 Dependencies

The *swap unit* GETREF is used in the *task* CHOOSE_GUIDE_STARS.

5.2.11.5 Interfaces

The GETREF *swap unit* is invoked thus: `return_status = execute (SWAP_GETREF);` where `#define execute load` is in effect in the flight version.

5.2.11.6 Resources

This *swap unit* reads/writes only a few words from/to *global RAM* requiring only 1 DPU cycle. It requires very little DSP RAM.

5.2.11.7 References

See XMM-OM/PENN/TC/0004.03.

5.2.12 `su_init_exp.c`

5.2.12.1 Type

C code written for the White DSP *swap unit* INIT_EXP.

5.2.12.2 Purpose and Function

This *swap unit* is called by the INIT_EXP *task*. It sets up the exposure id and sets up the full frame, half resolution data acquisition configuration for the Blue DSPs.

5.2.12.3 Subordinates

This module includes `setup.h`.

5.2.12.4 Dependencies

The *swap unit* INIT_EXP is used in the *task* INIT_EXP, which generally will be run before *tasks* ACQUIRE_FIELD and CHOOSE_GUIDE_STARS.

5.2.12.5 Interfaces

The INIT_EXP *swap unit* is invoked thus: `return_status = execute (SWAP_INIT_EXP);` where `#define execute load` is in effect in the flight version.

5.2.13 `su_initialize.c`

5.2.13.1 Type

C code written for the White DSP *swap unit* INITIALIZE.

5.2.13.2 Purpose and Function

This *swap unit* initializes all static variables and shared variables with their respective default values. It is used to reset all system parameters to working default value and zero out all storage memories. Some of the default values (*e.g.,* that of frames_per_exposure) may be overridden by command. An alert is sent to ICU after the completion, in the INITIALIZE *task*, signaling the readiness of the DPU for normal science operation. This *swap unit* will be run once at the beginning of, and not during, normal operation.

5.2.13.3 Subordinates

This module includes `setup.h`.

5.2.13.4 Dependencies

The *swap unit* INITIALIZE is used in the *task* INITIALIZE.

5.2.13.5 Interfaces

The INITIALIZE *swap unit* is invoked thus:
`return_status = execute (SWAP_INITIALIZE);`
where `#define execute load` is in effect in the flight version.

5.2.14 `su_locategs.c`

5.2.14.1 Type

C code written for the White DSP *swap unit* LOCATEGS.

5.2.14.2 Purpose and Function

This *swap unit*, which is used only in the TRACK_GUIDE_STARS *task*, calculates the locations of the *guide stars* in the most recently completed *tracking frame*. The *guide star* image is analyzed in the *tracking science windows*, based on the *guide stars'* last recorded location. This *swap unit* uses a simplified version of the image analysis algorithm derived from that used in `su_scanbs.c`, yielding only the centroid and brightness of the *guide stars*. The new locations of the *guide stars* are saved as the stars' current coordinates, to be used by the CALDRFT *swap unit*.

5.2.14.3 Subordinates

This module includes `setup.h`.

5.2.14.4 Dependencies

The *swap unit* LOCATEGS is used in the *task* TRACK_GUIDE_STARS.

5.2.14.5 Interfaces

The LOCATEGS *swap unit* is invoked thus:

```
return_status = execute (SWAP_LOCATEGS);
```

where `#define execute load` is in effect in the flight version.

5.2.14.6 Resources

This *swap unit* reads the following from *global RAM*: 258 words of window configuration data; about 576 words per star image for 10 stars. Thus, it reads a total of about 6000 words, requiring about 33 DPU cycles. It writes about 96 words if the guide star coordinates are saved, requiring an additional DPU cycle. For variables, it requires about 6000 words of DSP RAM.

5.2.14.7 References

See XMM-OM/PENN/TC/0004.03.

5.2.15 su_makeags_id.c

5.2.15.1 Type

C code written for the White DSP *swap unit* MAKEAGS_ID.

5.2.15.2 Purpose and Function

This *swap unit*, which is used only in the ACQUIRE_FIELD *task*, identifies the up-linked *absolute guide stars* in a list of bright stars generated from the field acquisition *reference frame* by the SCANBS *swap unit*. The identifications are made iteratively on the basis of position comparison and matching criteria.

An absolute offset search space is set up to correspond to the nominal spacecraft pointing error. It is subdivided into N $\times$ N search boxes, where N is given by the `SEARCH_SPACE_DIMENSION` parameter. Each search box represents an error range between the *absolute guide star* location and its identified bright star.

For each search box, the algorithm loops through each *absolute guide star* for identification in the bright star list. The brightest star in the bright star list whose offset relative to the *absolute guide star* under consideration is within the error box is taken to be the corresponding location. After all *absolute guide stars* are identified this *swap unit* exits and CALDRFT is invoked by the ACQUIRE_FIELD *task*. The calculated offset is then used to facilitate further identification with a more restricted error range. For each search box, an iteration is done to narrow the star identification criteria.

If the true *acquisition offset* is not consistent with the search box, then the identification-drift calculation iteration will lead to a small number of matches. If the true *acquisition offset* is consistent with the search box, then we expect to have a large fraction, if not all, of the *absolute guide stars* identified accurately: The number of matches between *absolute guide stars* and the bright star list are used as convergence criterion for the iteration. If no converged solution is found, then the *acquisition offset* is set to nil.

5.2.15.3 Subordinates

This module includes `setup.h`.

5.2.15.4 Dependencies

The *swap unit* MAKEAGS_ID is used in the *task* ACQUIRE_FIELD.

5.2.15.5 Interfaces

The MAKEAGS_ID *swap unit* is invoked thus:

```
return_status = execute (SWAP_MAKEAGS_ID);
```

where `#define execute load` is in effect in the flight version.

5.2.15.6 Resources

The number of words this *swap unit* reads from *global RAM* is about twice the sum of the number of reference stars and the number of bright stars, which has 1090 as a maximum value. It writes only a few words, and that only on the call that produces the final solution. Thus the total requirement is about 7 DPU cycles. It requires upto 1.1 kwords of DSP RAM. That number is sensitive to the value of `MAX_BRIGHT_STARS`.

5.2.15.7 References

See XMM-OM/PENN/TC/0021.01.

5.2.16 su_restore_stars.c

5.2.16.1 Type

C code written for the White DSP *swap unit* RESTORE_STARS.

5.2.16.2 Purpose and Function

This *swap unit* restores the coordinates of stars in a "catalog" to the unbinned detector pixel scale and uses polynomial approximations to the MIC distortion to compensate for spatial non-linearity. In the ACQUIRE_FIELD *task* it transforms bright star coordinates (as well as second moments and spatial integration ranges) from binned to unbinned pixel units and corrects the coordinates for the effect of spatial distortion. In the CHOOSE_GUIDE_STARS *task* it is first used to transform the bright star coordinates from binned to unbinned pixel units (without compensation for spatial distortion), and later used to correct the coordinates of the selected *guide stars* for the distortion effect. In the TRACK_GUIDE_STARS *task* it is used to correct the coordinates of the relocated *guide stars* for the distortion effect.

The parameter `RESTORATION_MODE` defines the order of the polynomial used to perform the correction. Supported values are 3, 4, 5, 6, and 7. The spatial distortion correction is not performed for any other value. As of this writing, the parameter is defined as 0 (zero).

5.2.16.3 Subordinates

This module includes `setup.h`.

5.2.16.4 Dependencies

The *swap unit* RESTORE_STARS is used in the *tasks* ACQUIRE_FIELD, CHOOSE_GUIDE_STARS, and TRACK_GUIDE_STARS.

5.2.16.5 Interfaces

The RESTORE_STARS *swap unit* is invoked thus:

```
return_status = execute (SWAP_RESTORE_STARS);
```

where `#define execute load` is in effect in the flight version.

5.2.16.6 Resources

Global RAM requirements for this *swap unit* depend on the job it is doing. If it is processing the `BRIGHT_STARS`, it requires about 20 times the number of bright stars (a maximum of 1024 words, or 4 DPU cycles). If it is processing `INITIAL_GUIDE_STAR_COORDS` or `CURRENT_GUIDE_STAR_COORDS`, it requires up to 64 words, or 1 DPU cycle. It requires about 200 words of DSP RAM.

5.2.16.7 References

See XMM-OM/MSSL/TC/0015 and XMM-OM/PENN/TC/0010.01.

5.2.16.8 Processing

5.2.16.9 Data

5.2.17 `su_rtrd.c`

5.2.17.1 Purpose and Function

This swap unit handles setup for and acquisition of engineering data. ENGINEERING - This swap unit sets up detector window parameters and writes them into PROC memory so that engineering data are written into known memory locations. For modes ENG_RAW_DATA, CENTROID_LUT_CALC, and ENG_PULSE_HEIGHT these window parameters do not correspond to detector pixels. In the engineering modes, the ICU dictates the portion of the detector to use; the output engineering data will be compatible with the window parameters defined here.

5.2.17.2 Subordinates

This module includes setup.h

5.2.18 `su_scanbs.c`

5.2.18.1 Type

C code written for the White DSP *swap unit* SCANBS.

5.2.18.2 Purpose and Function

This *swap unit* makes a catalog of bright stars in an image as follows:

- a *reference frame* is searched in raster fashion for pixels with more than a threshold number of counts and indices to those pixels are entered into a buffer;
- when the buffer becomes full, the neighborhood of each bright pixel is searched for the single brightest pixel;
- image properties of the bright spot (*e.g.,* counts, centroid position, second moments) are derived and recorded in the "bright star catalog," bright_stars[];
- an index array is created which can be used to access bright_stars[] in order of decreasing star brightness; and
- the indices are used to save the bright star catalog in *global memory* in this order;
- once all above-threshold pixels in the buffer are processed for bright stars, the buffer is reset and bright pixels above threshold is searched in the remaining portion of the reference frame. This procedure continues until the entire reference frame image is processed;
- the total number of bright stars found in the reference frame image is monitored. If there are too many bright stars, the threshold is raised and the scan redone. If there are too few bright stars, then the threshold is lowered and the scan redone.

SCANBS is used in the *tasks* ACQUIRE_FIELD and CHOOSE_GUIDE_STARS. In the latter the bright star catalog is searched for star concentrations (*i.e.,* crowded fields) and these are recorded for later use as exclusion zones when *guide stars* are sought for tracking in SCANGS.

In this *swap unit*, the detector field of view is mapped into `NMAXX` × `NMAXY` (nominally 1024 × 1024) "binned pixels." Hence the bright star coordinates and the crowded box locations are expressed in units of binned pixels.

5.2.18.3 Subordinates

This module includes `setup.h`.

5.2.18.4 Dependencies

The *swap unit* SCANBS is used in the *tasks* ACQUIRE_FIELD and CHOOSE_GUIDE_STARS.

5.2.18.5 Interfaces

The SCANBS *swap unit* is invoked thus: `return_status = execute (SWAP_SCANBS);` where `#define execute load` is in effect in the flight version.

5.2.18.6 Resources

This *swap unit* reads about
`5 + (# iterations) * ((nmaxx * nmaxy) + (nominal_x_dim * nominal_y_dim) * nbs)` words from *global RAM*. This is typically about 1.3 Mwords per iteration. Currently there is a limit of 4 iterations. It writes about `1 + nbs * SIZEOF_STAR` words to *global RAM*, plus a few more if a log is kept. This will typically be about 6.7 kwords if `nbs = max_bright_stars = 512`. In all, this amounts to about 6 seconds for I/O over the bus.

The DSP RAM requirements are: words for about 615 external and about 8044 internal variables, plus several more from auxiliary functions. Thus the total requirement will be about 8700 words if `MAX_BRIGHT_STARS = 512`.

5.2.18.7 References

See XMM-OM/PENN/TC/0004.03 or the most current version.

5.2.19 `su_scangs.c`

5.2.19.1 Type

C code written for the White DSP *swap unit* SCANGS.

5.2.19.2 Purpose and Function

This *swap unit* is used in the CHOOSE_GUIDE_STARS *task*.

This *swap unit* selects a list of good *guide stars* for tracking from the from a list of bright stars generated by the *swap unit* SCANBS. The *guide star* selection is based on the individual stellar quality indices and other global considerations. The star quality indices include factor such as the size, skewness, and brightness of the image. The global considerations are enforced to avoid situation such as *guide stars* in crowded regions, crowded *guide stars* in a local region, and *guide stars* too close to the edge of the field of view.

By default, `MAX_GUIDE_STARS` (16) are sought. Data are retained for as many as `MAX_REJECTS` (16) brightest objects which do not satisfy the *guide star* selection criteria, however these data are down-linked only if "verbose" reporting of Reference Frame Data has been selected (see Fig. 7).

This program yields 1) locations of guide stars in the current (reference) frame, 2) quality of guide stars, 3) counts in guide stars in the reference frame, 4) coords of "bright stars" that do not qualify as guide stars 5) quality of rejected stars, and 6) ref frame counts in rejects.

EXCLUDE_BORDER - EXCLUDE CROWDED FIELDS - EXCLUSION ZONE AND ADD EXCLUSION BOX -

5.2.19.3 Subordinates

This module includes `setup.h`.

5.2.19.4 Dependencies

The *swap unit* SCANGS is used in the *task* CHOOSE_GUIDE_STARS.

5.2.19.5 Interfaces

The SCANGS *swap unit* is invoked thus: `return_status = execute (SWAP_SCANGS);` where `#define execute load` is in effect in the flight version.

5.2.19.6 References

See XMM-OM/PENN/TC/0004.03 or the most recent version.

5.2.20 su_submit_compress_q.c

5.2.20.1 Type

C code written for the White DSP *swap unit* SUBMITCMPRSSQ.

5.2.20.2 Purpose and Function

This swap unit manages the timing of DPU operations, selects the task requested by the ICU, and queues data (i.e. science data, image, engineering image, and references frame) for delivery in the telemetry stream.

This *swap unit* is used in the FINISH_FRAME *task*. When it is determined that an *exposure* has completed and the *shift-and-add* of the final *tracking frame* into the complete *accumulated image* is done, this *swap unit* is called to place the `DD` data on the compression queue. [This may be made a default option in a future version, with the old calls to DELIVERDATA as the other option.] Data on the compression queue are compressed as part of the COMPRESS_DATA *task* by the COMPRESS *swap unit*.

BLUEFAST_DAT

5.2.20.3 Subordinates

This module includes `setup.h` and uses external program global_ram.c.

5.2.20.4 Dependencies

The *swap unit* SUBMITCMPRSSQ is used in the *task* FINISH_FRAME.

5.2.20.5 Interfaces

The SUBMITCMPRSSQ *swap unit* is invoked thus:
`return_status = execute (SWAP_SUBMITCMPRSSQ);`
where `#define execute load` is in effect in the flight version.

5.3 Red DSP Software

The Red DSP software is partitioned into two packages: the operating system known as RedOS; and the Red science software. The Red0S is mostly a subset of the DPUOS, containing all the functions to support the science software written in C. The symbolic layout of the Red DSP software is shown in Figure 11.

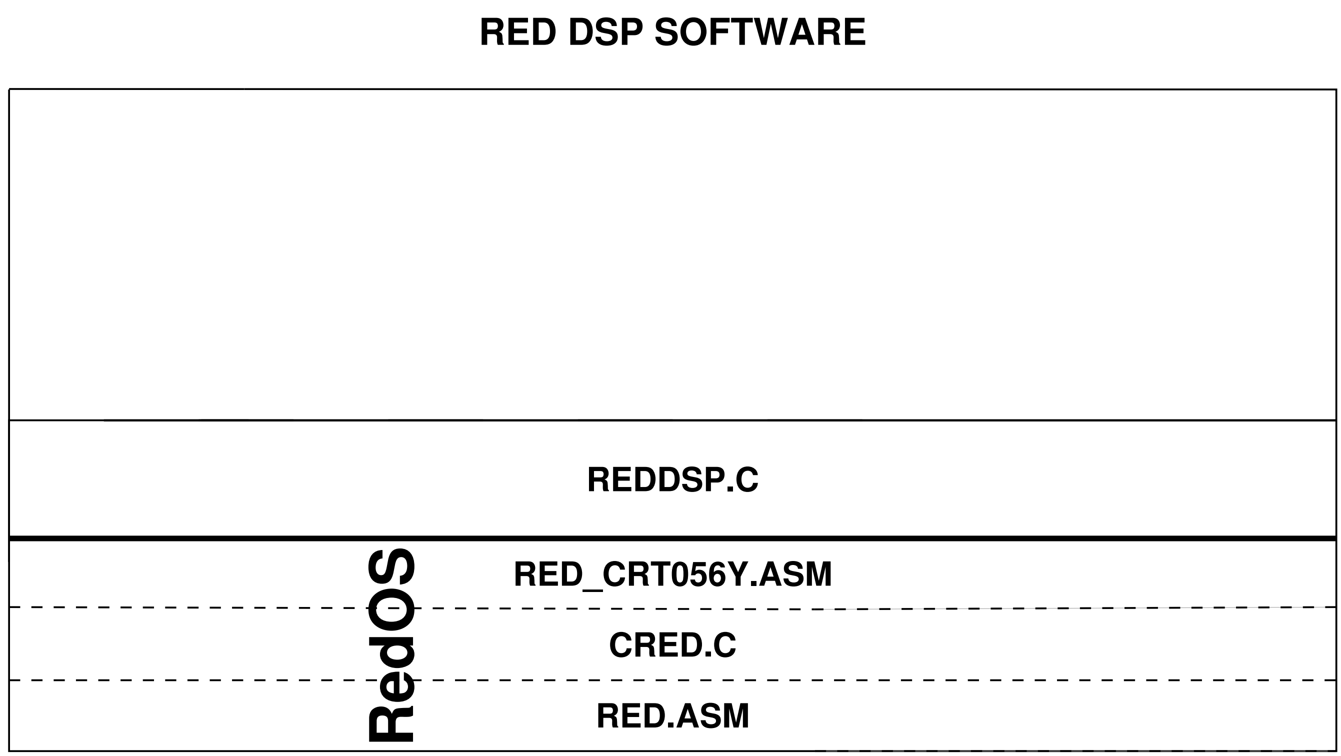

Figure 11: Symbolic layout of the Red DSP software. See text for details.

5.3.1 RedOS:red.asm

5.3.1.1 Type

Assembly routine running on Red DSP (part of RedOS).

5.3.1.2 Purpose and Function

```
1) Flight Loader Code
a) FecTab.asm
b) FecCode.asm
i) FfwrdErrCrection
2) xmmom_std.asm
3) hopper.asm
4) IN_DATA
a) Not used
5) Fprocess_data
a) Not Used
6) Fraw_data
a) Not Used
7) BUSIO
8) Fload
a) C-callable
b) Sets up the input of a swap unit of executable code and then blocks until the
Floader has the highest priority and is called by the BUSIO ISR.
9) Floader
a) Routine called by the BUSIO ISR to do the actual loading and FECing of the
requested swap unit.
10) FfecABlock
a) C-callable routine to scrub the Global Program Storage area of Single Event Upsets
i) Sets up the scrubbing and then blocks unitl it's completed.
ii) FEC scrub a block CODE.
11) ScrubLoader
a) Routine called by BUSIO to do the scrubbing of the Global Program Storage area.
12) Essitx
a) Unused stub.
13) Ssitx
a) Unused stub.
14) Ssirx
a) Unused stub.
15) Essirx
a) Unused stub.
16) ESCIRec
a) SCI Recevier interrupt w/error
17) SCIRec
a) SCI receiver interrupt
18) Fread_n_zero_sw_int_array
19) read_global
20) write_global
21) zero_global
22) Stkerr
a) Stack error interrupt
23) Trace
a) Trace  interrupt
24) Illi
```

```
a) Illegal interrupt
25) Swi
a) Software interrupt
26) F__start
a) INITIALIZATION
27) sci_command_inter
a) Called by the SCI receiver ISR to execute what ever command was received from the White DSP
```

5.3.2 RedOS:`red_crt056y.asm`

5.3.2.1 Type

Assembly routine running on Red DSP (part of RedOS).

5.3.2.2 Purpose and Function

A file that is used to describe the location of all global variables used by both the C and assembly code. It also defines the location of the stack, the size of usable memory, and a few other variables used by C memory management tools.

5.3.3 `cred.c`

5.3.3.1 Type

C routine running on Red DSP (part of RedOS).

5.3.3.2 Purpose and Function

This file contains basic Red DSP functions that are easiest implemented in C. The `main()` routine simply calls `reddsp()`, which is the master function of the Red science software.

5.3.3.3 Subordinates

The `cred.c` code includes `setup.h` and `cred.h`. Its `main()` routine simply calls `reddsp()`.

5.3.3.4 Processing

5.3.3.5 Data

5.3.4 Red Science Software:`reddsp.c`

5.3.4.1 Type

C routine running on Red DSP.

5.3.4.2 Purpose and Function

The file `reddsp.c` contains three parts: `reddsp()`, red task routines, and auxiliary routines.

The `reddsp()` routine is the traffic control routine for the Red DSP. It is simply an infinite loop continuously checking the dynamic variable `red_task_id`. When `red_task_id` changes (from `RED_IDLE_TASK`), `reddsp()` executes the appropriate task code. Unlike the White DSP code, the Red DSP code is small enough to be self-contained; there are no Red *swap units*. The following are currently supported *tasks*:

RED_IDLE -

This *task* is actually a "state" (see section 2.6.4). It does nothing but wait for the `red_task_id` variable to change. When `red_task_id` does change (by command from the White DSP via SCI link), the indicated *task* routine is invoked. Each of the Red DSP *tasks* resets `red_task_id` to `RED_IDLE_TASK` when finished, returning control to this state. There is no other code associated with this state.

RED_ABORT -

This task is a place holder for emergency exit. Currently it is executed only if an invalid `red_task_id` is seen in the RedOS–Idle loop. Its only action is to return control to the RedOS–Idle state.

RED_INITIALIZE -

This *task* is used only once, just after DPUOS is loaded. In fact, RED_INITIALIZE is commanded from the White DSP INITIALIZE *task* in the current implementation. Its purpose is to initialize all Red DSP parameters to default values. When done, it returns control to the RedOS–Idle state.

RED_ACCUMULATE_IMAGE -

This *task* performs the *shift-and-add* operation for all image mode *science windows* (mode = 0). Based on the calculated drift of the *tracking frame* being processed relative to the reference frame, the collected image in the image mode window is shifted and added onto an *accumulating image*, with appropriate binning (according to `param1` and `param2` of the *science window* configuration). Correction is made for translation offsets only (and not for roll). This *task* is straightforward but can be very time-consuming depending on the number of pixels involved. The *task* calls an optimized summation engine hand-coded in assembly. In the current implementation, the RED_ACCUMULATE_IMAGE *task* is uninterruptable. Upon completion of the *shift-and-add*, the *task* code sets a flag in PROC memory to inform the White DSP that the *shift-and-add* has been completed.

While the algorithm is trivial, care must be taken to incorporate the specialized memory access scheme in the DPU. With memory access limited and non-random, the algorithm is implemented in a read-a-chunk process-a-chunk fashion. The code is optimized such that it can read in an optimal amount which can be processed between available memory accesses. At the core of the *shift-and-add* code is a series of summation engines called `y_hopper()` coded in assembly language. To help expedite the processing, the optimal `y_hopper()` will be placed on the *on-chip memory* of the processor. For the simulation version of the codes, the `y_hopper` engines are coded in C.

If, as a result of drift by an amount greater than that anticipated when the windows were configured, a *science window* drifts beyond the edge of a *memory window*, some degradation will occur at the edge of the accumulating image. For this reason, the *science windows* should be embedded in *memory windows* surrounded by a border at least as wide as the estimated maximum drift in the *exposure* time (nominally 1000 s); in this case the pixel exposure will be uniform in all *science window* pixels (*i.e.,* same cumulative integration time per pixel). Here the word “exposure” is used in the meaning commonly used in X-ray astronomy: the effective area and duration that a part of the sky is imaged are referred to as exposure. The term “exposure map” is commonly used for imaging and as a normalization procedure before any further analysis is done. In principle, tracking history information can be used to create an exposure map for cases when a *science window* does drift out of its *memory window*.

5.3.4.3 Subordinates

The `reddsp.c` code includes files `setup.h` and `global_access.c`. When compiled for simulation on a Sun platform, it also includes `shared_variables.c`, `sun_util.c`, and `shared_io_functions.c`.

5.3.4.4 Dependencies

5.3.4.5 Interfaces

The Red DSP traffic control routine is called from the `main()` routine in module `cred.c` thus: `reddsp();`. In nominal operation it will never return to the calling routine.

5.3.4.6 Resources

The RED_ACCUMULATE_IMAGE *task* is the only significant `reddsp.c` user of resources, and has the following *global memory* requirements:
- reads 258 words of window configuration data
- reads N_pix words of *tracking frame* data, where N_pix is the total number of detector pixels in all image mode *science windows* (not the larger *memory windows*).

• reads and writes N_pix/B words,
where the pixel binning factor `B = (1 << bin_factor_x) * (1 << bin_factor_y)`.
Thus the total number of words is 258 + N_pix * (1 + 2/B). An upper limit on N_pix is implied by the fact that the bus access time cannot be greater than the *tracking frame time*, currently set at 10 seconds, but may increase to as high as 20 s. Assuming 10 second frame time, the maximum values for N_pix for the unbinned (1×1) and 2×2 binned cases are 850 000 and 1 700 000, respectively. This corresponds to upper limits on the number of DPU cycles of about 3320 and 6640. Note that computational overhead will increase the time it takes the *task* to complete. The *task* requires about 830 words of DSP RAM.

5.3.4.7 References

See XMM-OM/PENN/TC/0004.03 (or the most recent version).

5.4 Blue DSP Software

The Blue software runs on both Blue DSP 1 and Blue DSP 2, and consists of a single component which performs the initial processing of the Blue Detector data. This component is written in assembly language due to the processing speed required by the maximum Blue Detector count rate.

5.4.1 `blue.asm`

5.4.1.1 Type

Assembly routine running on both Blue DSPs.

5.4.1.2 Purpose and Function

1) Bootstrap Loader

a)Upon removal of the Reset signal (generated on the Arbiter board) the Blue processor executes the Bootstrap Loader to transfer executables from the "Blue" portion of the Global Program Storage RAM into the program execution area of the on board memory.

b) FecTab.asm: Routine that initializes the mask patterns that are used to generate/check the Forward Error Correction Word used to verify a block of op-codes.

c) FecCode.asm (FfwrdErrCrection) Routine that checks the validity of a block (15 24Bit op-codes followed by 1 24Bit Forward Error Correction Word) of op-codes. If a Single Bit Error is discovered it is corrected in the op-code that is currently being held in the on board RAM.

2) Start: System initialization routine that is pointed to by the reset vector. This routine configures all on chip and on board peripherals and initializes all software constants and variables. When completed with initializations it enables interrupts for data collection, global bus activity, and command reception.

3) CLC2: Main polling loop for address calculation and Fast Mode compression, and Raw data output. The loop is the background task that the processor spends all of it idle cycles in.

4) Calculate

a) Routine that converts the 24Bit event word received from OM1 via the data capture interface into an address and an increment value. The address points to the memory location in Small Word memory that represents the number of photons that this pixel has been illuminated with. The increment value, now always set to one by the Blue processing electronics, is used to increment the address in Small Word memory.

5) Calculate:FM This routine converts the counts, read from Small Word memory into an on board array, into the packed format used to telemeter the Fast Mode Data to the ground. Prior to the start of an observation a list of addresses, representing pixels in scientific areas of interest, are passed to the Blue processor along with a sample time. The sample time is the number of milliseconds between readings of the provided list of pixels. When the delay has been completed the processor reads the address, compresses the data in to the

Fast Mode Format, and stores that data in the Global RAM.

6) BUSIO: Global BUS Interrupt Service Routine. Global resources are accessed through this routine, invoked when the Arbiter card passes control of the Global Bus to this processor. This routine can be best described as a series of If-Then-Else statements. The number of transactions it performs with the global resources defines the amount of time a processor has the global bus. That number of transactions is fixed at compile time and care should be made when requesting reads or writes to global resources.

7) Read_FM_Pixels: Called from BUSIO, reads Small Word RAM locations that contain the values of the pixels that are in the current Fast Mode Window. Those locations are stored as an array of addresses that are passed to the processor via the Proc. area of Big Word RAM.

8) Dump_Eng_data

a) When commanded to, this routine outputs unprocessed 24Bit words that were received via the Data Capture Interface, note that the receiving hardware strips off the parity bit from the 24Bit word. Each 24Bit word is split into two 16 bit words, the first contains the most significant 8 bit, the second the least significant 16 bits.

9) ESCIRec: SCI Recevier interrupt w/error. This routine handles messages from the White processor that are incomplete or have framing problems. When an error has been detected it resets the interface and flushes the incoming command buffer.

10) SCIRec: SCI receiver interrupt. The SCI (Serial Communications Interface) receiver routine buffers bytes received over the SCI interface into 24Bit words. It keeps track of the number of bytes received until the number received is equal to the transmission length which is embedded in the second byte of the message. When the correct number of bytes have been received, the complete message is passed to the command interpreter for execution.

11) COMMAND_INTERPRETER

a) Called from SCIRec to execute the command received from the White processor via the SCI interface.

i) Commands: Heart beat request, Use Ping address space constants, Use Pong address space constants, Enable Klingon, Disable Klingon, Enable Events, Disable Events, Enable raw data output, Disable FM data output, Enable RAM test, Disable RAM test, Load image window constants, Load Fast Mode window constants, Enable FM data output, Echo ping address calculation constants, Echo pong address calculation constants

12) Grab_FM_Constants

a) Called from BUSIO, after enabled by SCI command “Load Fast Mode window constants”. This routine reads the variables and the array of addresses that define a fast mode window. Once all the information has been transferred from Big Word RAM to Onboard RAM the routine disables further calls from BUSIO.

13) Grab_Address_Constants

a) Called from BUSIO, after enabled by SCI command “Load image window constants”. This routine reads the constants that are used to convert the 24Bit word received from the Data Capture interface into an address in Small Word memory and an increment value. Once all the information has been transferred from Big Word RAM to Onboard RAM the routine disables further calls from BUSIO.

14) Mem_test

a) Called from BUSIO, after enabled by SCI command “Enable RAM test”. This routine increments the last 256 addresses on each of the memory boards three times, then disables itself.

15) Illegal Interrupt

a) Illegal interrupt

i) Non-maskable internal interrupt that is triggered if the processor performs an op-code fetch and does not receive a valid bit pattern. Code stub provided to cause reset of system if execution should ever reach this interrupt.

16) Swi

a) Software interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

17) Stkerr

a) Stack error interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

18) Trace

a) Trace interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

19) Ssi1

a) Ssi interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

20) Ssi2

a) Ssi interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

21) Ssi3

a) Ssi interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

22) Ssi4

a) Ssi interrupt: Unused interrupt. Code stub provided to cause reset of system if execution should ever reach this interrupt.

5.4.1.3 Subordinates

No subordinate components are called by `blue.asm`. `blue.asm` uses the header files `asm_dpucfg_map.h` and `asm_sci_com.h`.

5.5 Utilities

5.5.1 `blue_split.c`

5.5.1.1 Purpose and Function

This program creates 3 files for programming BOOTSTRAP proms for the BLUE and RED processors. It uses the files blue_boot.lod as an input and red_boot.lod as inputs. These files should only contain the "loader" portion of the Blue and Red .lod files.

5.5.1.2 Subordinates

This module includes stdio.h.

5.5.2 `boot_split.c`

5.5.2.1 Purpose and Function

This program uses as input file for address range 0x000 through 0x7fff and pads it as the right (?) format.

5.5.2.2 Subordinates

This module includes setup.h.

5.5.3 `cal_checksum.c`

5.5.3.1 Purpose and Function

This program calculates the checksum for each file.

5.5.3.2 Subordinates

This module includes stdio.h, stdlib.h, and string.h.

5.5.4 `cmnder.c`

5.5.4.1 Purpose and Function

This program converts s-record formatted PROM loads into xmm-om formatted command strings, in order to reprogram the on-board EEPROMs.

5.5.5 `create_loader_lut.c`

5.5.5.1 Purpose and Function

This program creates a table file "loader.lut.table" which is list of number of available address-length jump look up table, for white, blue, red swap units.

5.5.5.2 Subordinates

This module includes setup.h.

5.5.6 `sizetest.c`

5.5.6.1 Purpose and Function

This program is to test the size of existing su units verse the allocated EEPROM space; it complains if segments of a program are overlapped.

5.5.7 `srec.c`

5.5.7.1 Purpose and Function

This unit converts Motorola DSP load file records to S-record format. Srec takes as input a Motorola DSP absolute load file and produces byte-wide Motorola S-record files suitable for PROM burning. If no file is specified the standard input is read. The Motorola DSP START and END records are mapped into S0 and S7/S9 records respectively. All other DSP record types are mapped into S1 or S3-type records depending on the source processor.

5.5.8 `strip_lod_wfec.c`

5.5.8.1 Purpose and Function

The make file "make_p7_white_asm" strips the white.lod file of the white.asm code, and thus other files can be concatenated with it (using make_p7) into one promeable srecord file, p7. "strip_lod": converted from make_p7_etc to be used instead of the several existing versions of this program.

5.5.8.2 Subordinates

This module includes stdio.h, stdlib.h, and string.h.

6 Library of DPU software package

6.1 Header files

The functions, types and macros of DPU software are declared in the headers. A list of the header files with brief descriptions of their contents is listed in Table 1, and complementary descriptions are given below. The organization of these header files in the DPU software in Figure 12 shows the connection of these headers to the DPU code (White, Red, or Blue), and their subheader files (e.g., for the set up file, *setup.h*).

Header files, or include files, are used to simplify the software organization and reduce the software maintenance overhead. Most of the header files consist of symbolic constant definitions of parameters used in the software. These parameter values are set in one easy-to-find location, making changes in these parameters easy to perform. Also in the header files are macro and structure definitions.

Figure 12 shows the organization of the header files. The shaded boxes represent software packages, unshaded boxes represent the header files, and the arrows show the "include" statements alerting the compiler to include the header file in the compilation. The header file `setup.h` in turn includes all the header files used by the science software in both the White and Red DSPs.

Some of the header files are used in both the assembly language software and the C software. To ensure consistency between the two packages, and to simplify software maintenance, a UNIX shell script is used to translate the header files from C to assembly language syntax (see Appendix C). That script is executed automatically by the DSP compilation scripts, so there is no need to modify the assembly language versions manually. These header files are indicated in Figure 12 by the dash-lined boxes.

`asm_dpucfg_map.h` contains an address map of the PROC area of global memory, and uses subheader files of dpucfg_map.h, cwhite.c and setup.h.

`cwhite.h` is an include file for cwhite.c. It defines offset to data S.A., length of data, header S.A., length of header, pingpong, defines external data and their procedure definitions, global bus flags, and SSI data output.

`dpu_util.h` defines formats of address range, with comments on the standard header.

`dpucfg_map.h` contains an address map of the PROC area of global memory.

`idiot.h` has the definitions of interface between ICU and DPU protocol datatypes. Since DPU has been developed at LLAL and SNL, and ICU has been developed at MSSL, some of the convention data type definition were different to each other. This file provides official code translation of among ICU, DPU, and their interace. All definitions and header files for communications between ICU and DPU, between DPUOS and whitedsp, and between DPU codes and SDT codes are included.

`macro.h` defines function macros such as headers.

`mic_restore_params.h` gives parameter definitions used in *su_restore_stars.c*.

`our_own_types.h` defines usage of floats and data type used in DPU WHITE DSP code such as an interger, short or long.

`red_buffers.h` defines buffers that must be allocated first at modulus boundaries as "static", i.e. for the rest of the source file being compiled.

`red_control.h` is used to control RED unit by adding file header, updating shift-and-add on Red DSP, and numbering red tasks. This header includes setup.h.

`red_crt0_equ.h` is a header for crt056y.asm

`red_default.h` provides default values of parameters used in Red DSP code.

`red_os_variables.h` handles buffers, pointers and flags.

`red_shared_variables.h` includes default values of variables used in Red DSP task cod.

red_static_variables.h defines static variables for Red C code. All items in this file are consistent to all flight units, and are stored in a common static variable area defined in crt056y.asm. The file crt056y.asm requires this include file for formats. The variables are defined as external subroutine in shared_variables.h.

sci_com.h defines the SCI communication parameters between DSPs.

sci_ram.h defines RAM reserved locations for SCI interface in white.asm.

setup.h is a header to run swap code on DPU or in simulation mode. It has subheader files of idiot.h, our_own_types.h, stdio.h, sun_util.h, dpu_util.h, dpucfg_map.h, sci_com.h, white_control.h, white_default.h, red_control.h, red_default.h, macro.h, white_shared_variables.h, red_shared_variables.h, and mic_restore_params.h. macro.h, white_shared_variables.h, red_shared_variables.h, and mic_restore_params.h.

white_os_variables.h contains SNL handle buffers, pointers and flags.

white_shared_variables.h contains default values of variables used in White DSP swap unit code.

white_static_variables.h contains static variables for LANL/PSU Swap Unit C code. These items are common to all flight swap units and stored in a common static variable area defined in crt056y.asm. This file format is required for use in crt056y.asm. These variables are externally defined as extern in shared_variables.h.

6.2 Utilities

aouthdr.h contains values for the magic field in aout header.

coreaddr.h contains memory map header.

dspext.h contains DSP EXT - DSP commom object file format definition extensions.

filehdr.h is for file headers.

forErrorCorrTable.h contains Forward Error Correction Table.

linenum.h There is one line number entry for every "breakpointable" source line in a section. Line numbers are grouped on a per function basis; the first entry in a function grouping will have l_lnno = 0 and the place of physical address will be the symbol table index of the function name.

maout.h contains common object file format and file orignization.

reloc.h contains relocation types for all products and generics.

scnhdr.h The number of shared libraries in a .lib section in an absolute output file is put in the s_paddr field of the .lib section header, the following define allows it to be referenced as s_nlib.

storclas.h contains storage classes.

syms.h defines for "special" symbols.

File Name	Purpose
asm_dpucfg_map.h	ASM syntax version of dpucfg_map.h
asm_idiot.h	ASM syntax version of idiot.h
asm_red_control.h	ASM syntax version of red_control.h
asm_sci_com.h	ASM syntax version of sci_com.h
asm_white_control.h	ASM syntax version of white_control.h
cred.h	Extern declarations and symbolic constants used by cred.c
cwhite.h	Extern declarations and symbolic constants used by cwhite.c
dpu_util.h	Function prototypes of the functions written in assembly used by the C code (e.g., read_proc_int) when running on the DPU
dpucfg_map.h	Symbolic constants used as addresses of the PROC variables relative to the starting address of the DPU configuration table
global_access.c	C wrapper codes for memory access
idiot.h	ICU–DPU Interface Official Translation - symbolic constants used as data type identifiers for communication between ICU and DPU
macro.h	Macros used in the C code (i.e. max)
mic_restore_params.h	Symbolic constants used in su_restore_stars to correct for spatial distortion on the MIC
our_own_types.h	Definitions of data types used on DPU and structures
red.h	Symbolic constants used by red.asm. Definitions of addresses (e.g. default), bits of SCI, and port C.
red_buffers.h	Define storage for OS buffers in static variable area of *on-board memory*
red_control.h	Symbolic constants for control between RedOS and reddsp.c
red_crt0_equ.h	Symbolic constants used by ASM code embedded in cred.c
red_default.h	Default values of symbolic constants used in the Red C software
red_os_variables.h	Define storage of RedOS variables
red_shared_variables.h	Extern declarations of variables shared by the C and A SM codes (declared in red_static_variables.h)
red_static_variables.h	Define storage of variables shared by *swap units* and/or C & ASM codes and stored in the static variable area of *o n-board memory*
sci_com.h	Symbolic constants used for SCI commands for communica tion between DSPs
sci_ram.h	Define storage for SCI interface variables
setup.h	Master header file that includes header files required for processor the code is being compiled to run on (i.e., DPU or Sun)
sim_blue_local_ram.h	Symbolic constants used by Blue simulation softwar e on a Sun
sun_util.h	Same as dpu_util.h, but for simulations on a Sun
vbtwl.h	Variable Block Tiered Word Length scheme structural definitions
white.h	Symbolic constants used by white.asm
white_buffers.h	Define storage for OS buffers in static variable area of *on-board memory* that must be allocated first at modulus boundaries
white_control.h	Symbolic constants for control between DPUOS and whitedsp.c
white_crt0_equ.h	Symbolic constants used by ASM code embedded in cwhite.c
white_default.h	Default values of symbolic constants used in White DSP code.
white_os_variables.h	Define storage of DPUOS variables
white_shared_variables.h	Extern declarations of variables shared by *swap units* and/or C and ASM codes (declared in white_static_variables.h)
white_static_variables.h	Define storage of variables shared by *swap units* and/or C & ASM codes and stored in the static variable area of *on-board memory*

Table 1: Brief descriptions of the header files

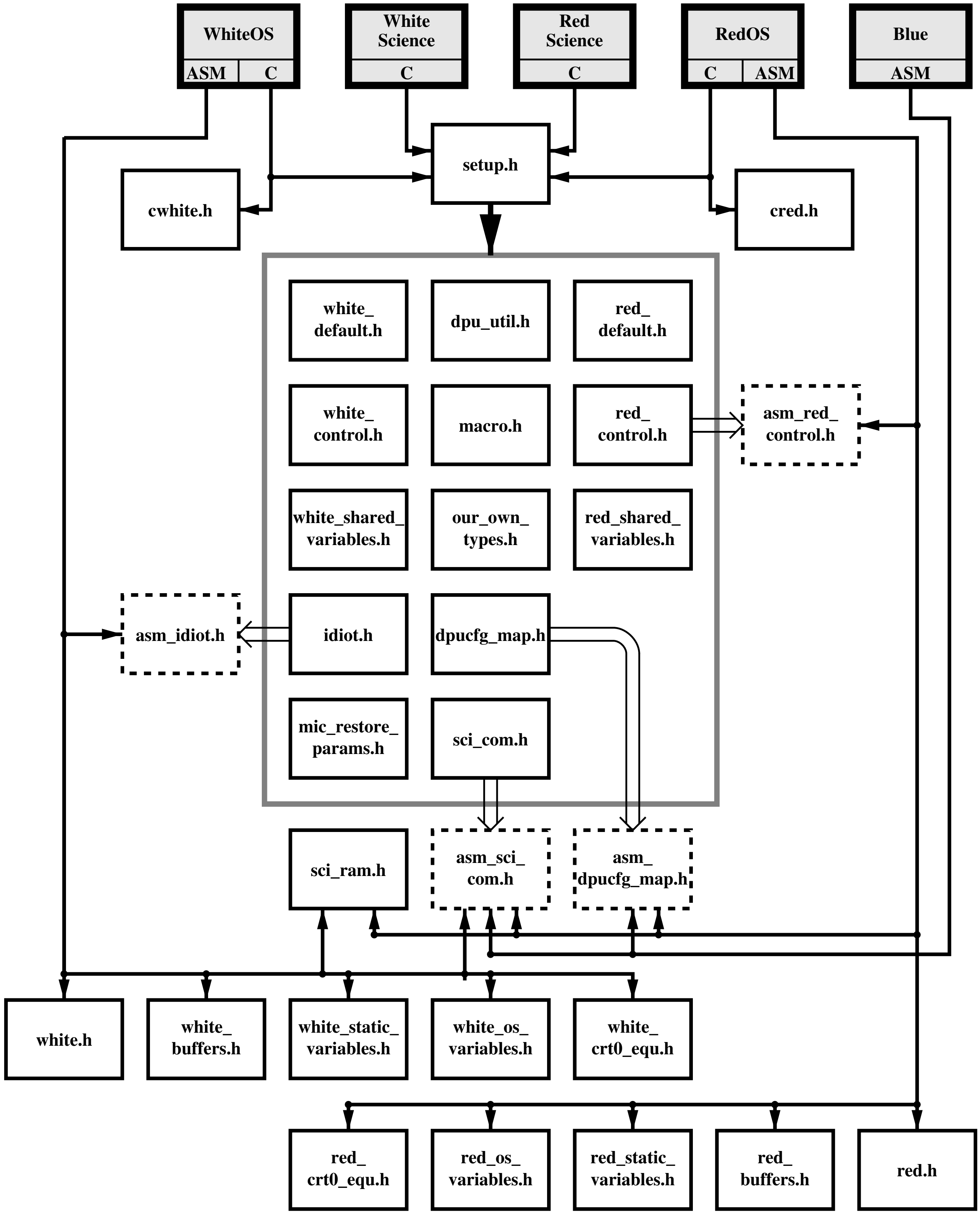

Figure 12: The organization of the header files in the DPU software, e.g. subheaders for *setup.h*. See text for details.

A Acronyms and Abbreviations

AD	Alert Data
ADD	Architectural Design Document
ANSI	American National Standards Institute
BDS	Blue Data Stream
BMW	Big Word Memory
BPE	Blue Processing Electronics (synonym for Detector Processing Electronics)
COTS	Commercially available Off The Shelf
CRC	Cyclic Redundancy Code
DBI	Digital Bus Interface
DBU	Digital Bus Unit
DEM	Digital Electronic Modules
DPU	Data Processing Unit
DSP	Digital Signal Processor
EGSE	Electronic Ground Support Equipment
EOB	Electro-Optical Model
FIFO	First-In First-Out
GNU	Gnu Not UNIX
GSE	Ground Support Equipment
GUI	Graphical User Interface
ICB	Instrument Control Bus
ICU	Instrument Control Unit
IDL	Interactive Data Language
KAL	Keep Alive line
LSB	Least SIgnifiant Bit
MIC	Micro-channel plate Intensified CCD
MSSL	Mullard Space Science Laboratory
MSB	Most Significant Bit
OBDH	On-Board Data Handling
OCI	Observation Configuration Information
OM	Optical Monitor
OS	Operating System
RAM	Random Access Memory
ROM	Read Only Memory
PC	Printed Circuit
PD	Priority Data
PROM	Program Memory
PSU	Power Supply Unit
RBI	Remote Bus Interface
RTRD	Real Time Raw Data Mode
SCI	Serial Communications Interface
SDE	Software Development Environment
SDT	Science Data Terminal
SSI	Serial Synchronous Interface (between the DPU and the ICU)
SWM	Small Word Memory
TM	Telescope Module
TSC	Telescope Simulator Card
VME	Versa Module Europa
X	X window system
^{XMM}OM	X-ray Multi-mirror Mission Optical Monitor

B ^{XMM}OM DPU Lexicon

Pointing — a period of time during which the spacecraft is locked onto a celestial field-of-view. A pointing can consist of one or more OM observations.

Observation — consists of one or more exposure(s) planned by one observer or group of observers. There can be more than one observation per pointing if there are other observational programs to be carried out on the same field-of-view.

Configuration — a instrument set-up during which the filter wheel position, window settings, etc. are fixed.

Exposure — the final collected data, consisting of Image Mode and/or Fast Mode data, and other data products associated with a single OM configuration. An OM exposure can consist of Image Mode data, $m \geq 1$ Image Mode frames co-added to produce the accumulated image (shift-and-add), and/or Fast Mode data, data from a sequence of Fast Mode time slices, as well as other data products. Each exposure is assigned a unique exposure identification number.

Exposure time (t_E) — an integer number of tracking frame times, $t_E = mt_I$, the time over which Image Mode frames are co-added to produce an accumulated image and/or the time over which Fast Mode data are acquired (typically $t_E \approx 1000$ s).

Frame — a single image collected at some interim point in an exposure by the DPU, which then operates on it for some identifiable task. This can be a tracking frame, Image Mode frame, or Fast Mode frame.

Tracking frame time (t_I) — the time over which events are accumulated in *Ping/Pong* memory to form a tracking frame. The frame time is constrained by the time required by the DPU to calculate a tracking solution and perform the shift-and-add. This is not a feature of the detector but of the DPU, and is not user selectable ($t_I \approx 20$ s [TBD]) but may be selected by the SOC (TBD).

Tracking frame — raw image collected in a tracking frame time stored in either the *Ping* or *Pong* memory. The tracking frame consists of photon counts for all detector pixels within memory windows, whether they are user requested or designated for guide stars.

Current tracking frame — is stored in either the *Ping* or *Pong* memory, and is where the received MIC events are registered by the Blue DSPs. This is also the location from where Fast Mode data are extracted.

Previous tracking frame — is stored in either the *Ping* or *Pong* memory — the opposite memory from where the current tracking frame is stored — and is used for calculating the tracking and is shifted and co-added to the accumulated image by the shift-and-add process. The physical location of the current tracking frame and the previous tracking frame are alternated every tracking frame time.

Image Mode frame ($I_{n;x,y}$) — a sub-area of the tracking frame consisting of just the detector pixels within the

Image Mode science windows. This frame is shifted and co-added to the accumulation frame by the shift-and-add process.

Accumulated image ($A_{x,y}$) — the Image Mode image stored in big-word memory, consisting of the Image Mode frames co-added with spacecraft drift compensation (result of the shift-and-add process).

Fast Mode slice period (t_F) — the time over which data are sampled and formatted in the Fast Mode frame out of the current tracking frame. This is user selected to be an integer number of MIC CCD frame times ($t_F = nt_{CCD}$), which is generally NOT an integer factor of the tracking frame time. The CCD frame time is dependent on the number of CCD rows read out (see description below). The MIC detector operates on a clock independent of the DPU clock. So, to avoid beating effects, fast mode synchronizes to CCD frame markers supplied by the MIC detector (see section 3.2 of the DPU ADD for details).

Fast Mode frame ($F_{m,n;x,y}$) — the image accumulated in the current tracking frame after a integer number of Fast Mode slice periods. Extracted directly from the tracking frame, this frame contains all the events accumulated from the beginning of the current tracking frame up to the current time; processing is required to subtract the events that occured in prior Fast Mode slices to obtain the Fast Mode time slice. A Fast Mode frame is limited to 512 detector pixels (a Fast Mode set). Each exposure can support up to two independent Fast Mode sets. Here m is the slice number, n is the tracking frame number, and x and y represent the spatial coordinates on the detector.

Fast Mode time slice ($S_{i;x,y}$) — the image from the Fast Mode frame collected during a Fast Mode slice period. A Fast Mode time slice is calculated by subtracting the previous Fast Mode frame from the current Fast Mode frame ($S_{i;x,y} = F_{m,n;x,y} - F_{m-1,n;x,y}$). A Fast Mode time slice is limited to 512 detector pixels. Because the Fast Mode slice period is short, < 1s, each Fast Mode time slice will consist of zero or a few events. Since they will be sparse, the Fast Mode time slices will be transmitted in differential address format rather than in image format.

Differential address format — a format for sparsely populated spatial and time data based on the differences between event addresses in a 1-D representation of the 3-D data space. Since Fast Mode data consist of few or no events per time slice, it is inefficient to transmit the time slice in image format. Instead, a list of pixels offsets (the number of pixels) between events is transmitted. For example, if in a Fast Mode area consisting of 512 pixels, the first time slice contains two events, one at pixel 36 and the other at pixel 352, the second time slice contains no events, and the third time slice contains one event at pixel 352, the DPU will transmit the numbers $36, 316(= 352 - 36)$, and $1024(= 512 + 512)$.

CCD frame — one readout image by the CCD detector in the MIC system. The CCD in MIC is read out nominally at a rate of about 100 Hz. This readout image is processed, in real time, for photon events by blue detector electronics. Addresses of significant events are sent to the DPU for data collection. The blue detector electronics will issue 2 (TBC) beginning of frame markers at the start of each new CCD frame. The DPU will use these marker to indicate the end of the previous frame and synchronize fast mode with the MIC sampling cadence. Two markers are required, one for each of the Blue DSPs collecting data from the blue detector.

CCD frame time (t_{ccd}) — the time between the readouts of the MIC CCD. It is nominally $t_{ccd} = 10\,\mathrm{ms}$, but could be shorter in the case of detector window configuration requiring only a small number of CCD rows. This defines the fundamental time resolution of the entire MIC system.

Global memory — DPU memory shared by all four DPU DSPs. Global memory is subdivided into small-word memory and big-word memory.

Small-word memory — global memory with 16 bit word length. There are 4194304 words of small-word memory on the DPU.

Big-word memory — global memory with 24 bit word length. There are 1048576 words of big-word memory on the DPU. Of this, 32768 words are reserved for PROC memory, leaving 1015808 for exposure data storage.

PROC memory — a sub-area of big-word global memory used for storage of data processing parameters and for inter-DSP communication purposes. There are 32768 words of PROC memory available.

$Ping/Pong$ memory — areas in small-word memory used to store tracking frames. While the current tracking frame is being accumulated, for example, in $Ping$, the previous tracking frame in $Pong$ may be used to determine the spacecraft drift (tracking) and the Image Mode sub-area co-added into the accumulated image (shift-and-add). The roles of the $Ping$ and $Pong$ areas are reversed after each tracking frame time.

Current exposure memory — areas of memory, consisting of both large-word and small-word memory, used for storing the transmitted data products of the current exposure while the exposure is in progress.

Previous exposure memory — areas of memory, consisting of both large-word and small-word memory, used for storing the to-be-transmitted data products of the previous exposure while the ICU is transmitting these data to the spacecraft. The physical location of the current exposure memory and the previous exposure memory are alternated after each exposure.

Local memory — memory located on the processor boards of the DPU, including the memory within the DSP (both on-board and on-processor memory). This memory area is accessible only by the one DSP physically located on the board where the memory is located.

On-board memory — memory located on the processor boards of the DPU, but external to the DSPs. This amounts to 32 kwords of 24 bit words of static RAM.

On-processor memory — memory located internal to the DSP56001. See DSP56001 User's Manual for details.

Detector pixel — a basic resolution element of the detector (for the MIC detector, a detector pixel is 1/8 of a CCD pixel [a.k.a. a centroided pixel]).

Science window (SCW) — a contiguous rectangular detector area in which scientific data are accumulated. There are three types of science windows: Image Mode, Fast Mode, and tracking. Each science window must fully reside within a memory window. There is no mutual exclusivity among science windows. No more than 16 science windows are available per exposure ($SCW \leq 16$), 10 of which are reserved as tracking science windows for tracking guide stars.

Image Mode science windows — user defined science windows for Image Mode data. Image Mode science windows are binned according to the user specified binning factors and accumulated over the exposure time. The Image Mode science windows will be slightly smaller than the memory windows to eliminate detector window edge effects of the shift-and-add process and allow for the acquisition offset shift.

Fast Mode science windows — user defined science windows for Fast Mode data. Fast Mode science window data are converted into Fast Mode time slices in the DPU and transmitted in differential address format.

Fast Mode set — a set of detector pixels consisting of one or more Fast Mode science window(s), consisting of no more than 512 detector pixels total. There may be up to 2 Fast Mode sets per exposure, with independent Fast Mode slice periods.

Tracking science windows — DPU defined science windows for guide stars. The size of tracking science windows is set in the DPU software to 32 $\times$ 32 detector pixels (tracking memory windows are 64 $\times$ 64 detector pixels). Tracking science window images are not transmitted, but data products relating to these windows are available.

Memory window (MMW) — a contiguous rectangular detector area stored in memory. The total memory window area cannot exceed 1048576 (1024^2) detector pixels, and the number of memory windows cannot exceed 16 ($MMW \leq 16$). Of the 16 possible memory windows, 10 (TBC) are reserved for the tracking science windows. Memory windows cannot overlap. Due to the size limitation of MIC Detector windows, some memory windows may consist of more than one detector window. They also must consist of entire MIC CCD pixels, thus only every 16th detector pixel can be at a memory window boundary and the dimensions of the memory windows must be divisible by 16. Because the MIC Detector centroiding algorithm uses a 3 $\times$ 3 MIC CCD pixel area, the minimum size of a memory window is 32 detector pixels, or 4 MIC CCD pixels, in both dimensions.

Detector window (DTW) — a contiguous rectangular area of detector pixels. MIC Detector detector windows are limited to 512 detector pixels in either dimension. No more than 15 detector windows can be assigned at a time on the MIC Detector ($DTW \leq 15$). Detector windows must consist of entire MIC CCD pixels, thus only every 16th detector pixel can be at a detector window boundary and the dimensions of the detector windows must be divisible by 16. Because the MIC Detector centroiding algorithm uses a 3 $\times$ 3 MIC CCD pixel area, the minimum size of a detector window is 32 detector pixels, or 4 MIC CCD pixels, in either dimension.

Tracking — the determination of the spacecraft attitude relative to the start of the exposure by comparison between locations of guide stars in the just-completed tracking frame and the reference frame.

Reference frame — the frame used as the star zero-drift position reference for the tracking process. The reference frame is a frame taken at the beginning of an exposure using the same filter wheel position as used for the scientific data.

Shift-and-add — the shifting of the Image Mode frame by the amount determined by the tracking to compensate for the spacecraft drift, and the co-adding of the frame to the accumulating image.

Guide stars — the stars used in the tracking process to determine the spacecraft drift by comparing their positions in the previous tracking frame to their positions in the reference frame.

Acquisition frame — the frame taken, at the beginning of a pointing using the V filter, for comparison with the up-linked catalog of predicted absolute guide star positions in order to determine the true pointing of the OM.

Absolute guide stars — the stars used in the acquisition process, at the beginning of a pointing, to determine the true pointing of the OM. The catalog of absolute guide stars is up-linked from the ground station for each pointing.

Acquisition offset — the relative attitude offset (pitch, yaw, roll) between the commanded and actual satellite pointing, calculated by comparison of the absolute guide stars' up-linked positions and observed positions.

Task — A sequence of procedures performed by the DPU's White DSP on command from the ICU.

Bright star submode — the DPU calculates the position and intensity of the 10 (TBD) guide stars for each of the tracking frames.

Transient submode — the DPU calculates the position and intensity of the 10 (TBD) brightest sources in the defined memory windows for each of the tracking frames.

C Compiling Utilities

The DPU code can be compiled for use with the DPU hardware or for simulation on a Sun workstation. The two versions use mostly identical high-level code, but must be compiled differently. The flight version must be compiled with the `g56k` compiler, while the simulation code may be compiled with any ANSI standard C compiler. The flight code is compiled on a Sun with a suite of makefiles, while a short shell script, which makes use of a single makefile, is all that is required for the simulation version. The simulation version of the "color codes" and several programs used in generating simulated data (which are used in ground tests of the DPU hardware and flight code) and their use are described in "DPU Simulation User's Guide" (XMM-OM/PENN/ML/0002) and will not be repeated here.