XMM Optical Monitor

DPU Simulation User's Manual

XMM-OM/PENN/ML/0002.02

Alan D. Welty
The Pennsylvania State University

and

Cheng Ho
Los Alamos National Laboratory

May 17, 1996

Department of Astronomy and Astrophysics

Contents

1 Introduction

This document explains the purpose of, gives descriptions of, and provides instructions for installation and execution of the ^{XMM}OM DPU simulation software. That software can be run in full simulation mode on a Sun/UNIX system or on the DPU itself. In the latter case, simulated data must be provided. There are differences between Sun and DPU implementation due to limitations of the former. Nevertheless, the Sun simulation code is designed to mimic real DPU operation as closely as possible.

Much of this document is based on notes compiled by Cheng Ho in 1993. The first half of those notes were transformed into Section 3 of "X-ray Multi-mirror Mission Optical Monitor Digital Processing Unit Electronic Ground Support Equipment and Software Development Environment" (DPU EGSE & SDE; document XMM-OM/PENN/SP/0005.01). It was thought that an update of those materials should be presented here, along with instructions for installation and use of the software, due to the evolving state of the simulation scheme and details of the mission itself.

This overview describes the simulation scheme for the ^{XMM}OM's software and hardware development, testing, and verification. Because a significant fraction of the project involves on-board data processing controlled by elaborate software on sophisticated hardware, a reasonably realistic simulation for most aspects of the system is essential to help design, implement, and test individual components. It is also needed to aid in development of diagnostic tools during testing, and analysis tools for processing the data products.

The simulation consists of two distinct parts: the data chain and actual DPU simulation. The former involves the simulation of the elements in the data chain in front of the DPU. Simulation of those elements outputs data suitable for input to the DPU. Such data can be input to the real DPU or the DPU simulation programs. The availability of test data with correct format and proper realism is critical in all phases of the testing, not only for software development, but also for processing, hardware debugging, and instrument diagnostics. The data chain scheme has the following characteristics:

1. It uses a modular approach. Although somewhat inefficient, the modular approach allows us to adjust the degree of realism for the simulation with relatively little effort.

2. This scheme is data driven. Since OM's main job is to detect photons and turn them into usable digital signals, it can be reasonably well simulated by a single chain of data filters. Specific, the main task of the simulation is to transform information of the some entity through an entire system. In the case of OM, this might be the information content of an input photon transformed from a ray to an analog signal to a digital signal and finally to a MIC event record. Each phase would correspond to a distinct component of the real physical situation. Each component in the simulation chain is simply a filter that modifies the information received from the previous component and sends the information to the next component.

3. The current scheme was developed under the UNIX operating system and utilizes its piping and multitasking capabilities. Since the OM consortium has largely converged on the Sun platform, this implementation should not be a problem. However, the philosophy should be easily ported to other operating system such as VMS. For example, a previous incarnation of the simulation used files to transfer information from one module to the next, a scheme easily accommodated by VMS. In the current scheme we have the freedom to break the chain at any point and collect the data into a file. The chain can easily be restarted by dumping the contents of the file to the break point of the chain. This will be discussed later. This ability to break the chain provides a great flexibility for diagnostics, and has been used extensively in DPU code development and testing.

4. The philosophy is to have each module represent a major component in the data chain. The interface between software modules should exactly mimic the interface between hardware modules as far as data representation is concerned. In this way we can swap simulation software and actual hardware with essentially no modification.

These philosophical points should become clearer with the examples below. Section 2 discusses the data chain simulation.

The DPU portion of the simulation code simulates the functions of the DPU on the UNIX workstation using a combination of the actual DPU flight software and special simulation software to perform the functions handled by the assembly level software on the DPU. The DPU simulation has the following characteristics:

1. The simulated Blue DSP, which is coded in C, carries out the MIC event to address calculation and image incrementation.
2. The high-level White DSP simulation codes are the actual flight codes compiled on the Sun using the `gcc` (Gnu C) compiler. We use the Motorola DSP 56001 compiler, **`g56k`**, to compile the White DSP code into DSP 56001 machine codes that can be run on the DPU itself. There are slight differences in the code for the two cases. C preprocessor directives are used to control this.
3. The global memory on the DPU is simulated by a file of 5M words. On the Sun, each word is 4 bytes. Thus the global RAM requires a 20 MB file. Three files containing essential information for control and communication for the Blue, Red, and White processors are used to simulate those portions of the local RAM.
4. A rudimentary ICU/DPU command processor simulator manipulates the RAM files to simulate commanding the DPU.

Section 2 of this document describes the philosophy and implementation of the simulation scheme we have developed.

Section 3 discusses the DPU simulation in more detail.

Section 4 briefly discusses the purpose of each of the simulation scripts and programs.

Section 5 provides instructions on how to install the simulation software.

Section 6 gives instructions on how to run the simulation software.

The Appendix provides current listings, with comments, of the main simulation scripts.

2 Simulation of Data Production and Event Processing

2.1 Turning Photons into Electrical Signals

The OM detects photons and converts some of their information content into electrical signals. The following points must be considered:

1. Photons leave the scene, i.e., field of view (FOV), toward the observatory. For our purposes, they are well represented by rays specified by wavelength, position, direction, and time (since we count photons with accurate timing). All of these quantities are stored as double precision numbers. The current implementation also includes a number called statistical weight.
2. Each photon may experience radiative transfer effects as they propagate to the observatory. For example, ground-based systems have to worry about atmospheric scattering. Space-borne astronomical systems such as OM probably do not have to worry about this. The output data are still rays.
3. Each photon is collected through some collecting device, i.e., telescope. The photon is now treated as a ray presented to the front of the detector at the focal plane.
4. Motion of the spacecraft (e.g., drift, roll) during an exposure must be taken into account when determining position of a ray in the focal plane.
5. The detector, will detect photons with finite quantum efficiency. The quantum efficiency may be simulated by Russian Roulette. Detected photons will be turned into analog signal also represented by a set of double precision numbers.

6. The detector electronics convert the analog signals into digitized signals in the defined MIC 24 bit event record format.
7. Further downstream there are usually data acquisition electronics which store and process the data. In principle there could be multiple stages in an electronics chain.

To summarize, a generic modern photon detection system will contain the following components in the data chain:

scene | transfer | platform | optics | detector | electronics

In the simulation each component is represented by a standalone software module. The vertical bar "|" is used to separate connecting modules. In UNIX it actually has a special meaning to be discussed below. Each interface from one module to the next should contain sufficient information for all downstream processing. Except for the first component, the scene, all components should be looked at as filters. The first component is usually a Monte Carlo simulator that generates photons represented by rays. The last component will generate data that can be collected from the standard output device or sent to a user specified file.

2.2 Module Composition

The software simulation module chain should follow the data flow. The exact breakdown of components depends on the circumstances, with the fundamental requirement that the software interface must occur at an identifiable physical or hardware interface.

For example, there is nothing to prevent us from breaking up the optics simulator for $^{\rm XMM}$OM into the following chain:

primary | secondary | dichroic | filter

For optical design specialists, this chain is easily recognized as a ray tracing exercise. We can also break up the detector, if we are simulating MIC, into the following components:

window | photocathode | MCP | taper | CCD | MIC electronics

As long as the data interface is kept the same going in and coming out, there will be no problem for the simulation user in making substitutions.

One of the shortcomings of this whole scheme is that data cannot flow back. For example, in the breakdown of the optics simulation (ray tracing) above, suppose a photon reflects from primary, scatters from part of the telescope structure and hits the primary again. This scheme will not be able to accommodate that situation. If this is a serious problem or if this is the effect one wishes to examine, then one should have a conglomerate optics module which carries out a full-blown ray tracing and Monte Carlo if necessary. For OM this particular example probably will not arise if baffling and light-tightening are properly designed and implemented.

The key point is that the interfaces should remain fixed when one does a break-up or merging of components. The interfaces should either carry sufficient information (double precision floating point numbers are preferred) if they are simulating physical devices, or adhere to defined digital interfaces if the information is digital.

2.3 Module Substitution

As long as we have a fixed interface, it is trivial to substitute one simulation module with another containing a different degree of realism. Using the optics module as an example, we could have a full blown ray tracing program, `optics0`, that includes realistic geometry, absorption, scattering, diffraction and refraction. We can also have a module called `optics1` that is a simple idealistic optics system which simply translates the directional vector of a ray into a position at the front of the detector. We can have a module, `optics2`, that applies a simple Gaussian representing the PSF of the optics. We can have a another module, `optics3`, that reads a file and interpolates the PSF from a tabulated spot diagram. There are many other variations in module realism that can be imagined and implemented.

As long as the interface for the input and output remains the same, both rays in this case, there is no problem substituting `optics0` with `optics1` or `optics2` or `optics3`. The only difference is the efficiency of the simulation. It may take much longer to run `optics0` than `optics1`. As a result, one might run `optics0` once or twice per year, but run `optics1` virtually every day. This will depend on what aspect of the data realism one wishes to simulate.

This scheme is also conducive to evolutionary build-up of the realism. We do not need to have a full-blown simulation, e.g., `optics0`, right from the beginning while we are still hashing out the optical design. We could start from `optics1`, build up to `optics2` and `optics3` and other possible steps before moving to `optics0`. At every step there will continue to be a working version. In contrast, an all-in-one simulator will be more difficult to manage since it would probably require major structural surgery each time one wishes to change the code to enhance the simulation algorithm.

2.4 UNIX Implementation

UNIX is a *multitasking* operating system (OS). This means that users can execute more than one process at a time. The OS takes care of scheduling task access to system resources automatically. This is transparent to users.

There are also system entities called *stdin* and *stdout*, standing for standard input and standard output devices. By default they are the keyboard and monitor, respectively. However, users can reassign both to other I/O devices, files, commands, or programs on the system.

This I/O redirection capability can make use of the UNIX piping capability. A UNIX pipe is represented by the "|" character. The UNIX command `a | b` means that output from `a` sent to *stdout* will be treated as input to *stdin* of `b`, completely transparent to the user. Multitasking means that both `a` and `b` will be resident (in RAM or virtual memory) until the pipe is broken. There is not much inefficiency when the OS executes different codes. Thus in UNIX, with "%" as a generic UNIX prompt, if we issue the command

```
% scene | optics | mic | bluedsp
```

to the OS, the string of commands will continue to execute until the pipe is broken or until all elements of the chain have completed. Typically the `optics`, `mic`, and `bluedsp` filters will run in an infinite loop, accepting data from *stdin* and sending data out to *stdout* until the `scene` simulator terminates at a preset time. It is also possible, for example, for `bluedsp` to have a predetermined termination time, at which it exits and brings down the entire pipe chain.

2.5 The Data Chain

As of this writing, software modules have been established for each component of the following data chain:

SCENE | TRANSFER | PLATFORM | OPTICS | DETECTOR | ELECTRONICS

Note that TRANSFER is not necessary for the $^{\mathrm{XMM}}$OM, a space project. The individual data chain programs are described in Section 4.

With this type of data chain scheme, we may run simulations with many practical variations. For example, we can examine the data at any point in the chain by replacing a pipe with a redirection of output to a file. For example, a pair of commands like

```
% scene | optics | mic > data.dump
% cat data.dump | bluedsp
```

will duplicate the result of the previous example, but saves data output by `mic`, the DETECTOR element of the data chain, as valid 24 bit MIC event records in a file called `data.dump`. In practice this is exactly what we do. Saving data is desirable because it allows us to run different versions of the simulation code mimicking the same functionality. (This is essential to maintain backward compatibility when we upgrade the simulation code.) The `data.dump` file could also be taken to the DPU EGSE and input to the real Blue DSP. That is, we can test the real Blue DSP on the DPU hardware with a well-defined and reproducible data set. This allows parallel testing and comparison of real and simulated Blue DSP.

3 DPU Simulation

3.1 DPU Overview

This is a brief description of the DPU hardware, processing task assignment and system software, as it relates to the high-level on-board data processing development and simulation. This is not meant to be a thorough review of the DPU. Consult the relevant documents for more details.

The DPU consists of 4 Motorola 56001 DSP processors running at 20 MHz. Each one resides on a board plugged into the back plane of the DPU. On each processor board there will be 32 K words of memory. A good analogy of this architecture is that we have 4 computers, each with some cache memory (on-chip) and some RAM (on-board local RAM). Accessing the cache memory is *not* explicitly done by any high-level White DSP codes.

For bulk storage, there is a total of 5M words of memory, called the global RAM. The global RAM is better viewed as a common "disk" storage shared among the four computers. Since the global RAM is accessed by all four computers via the same bus, a round-robin scheme is adopted to facilitate access and avoid collision. A very careful management of the memory access is required. The 5M words of memory are divided into two kinds, 1M words with dynamic range of 24 bits and 4M words with dynamic range of 16 bits. From the processors' standpoint, all words are 24 bits. It is the responsibility of the processing code to ensure that the appropriate type of memory is assigned for each usage.

The four computers are called Blue DSP 1, Blue DSP 2, Red DSP, and White DSP. The Blue DSPs will be heavily involved in data acquisition from the MIC detector. The Red DSP will be used solely for performing the shift-and-add calculations. The rest of the high-level data processing is carried out by the White DSP. All processing on the Blue DSPs is done with simple algorithm coded in assembly. There is no "operating system" on the Blue DSPs. On the other hand, the White DSP is responsible for high-level processing, communication with the ICU, and controlling of the Blue and Red DSPs. It requires an elaborate operating system (DPUOS) to handle real time interrupts and data crunching. The data processing flight codes rely on the DPUOS for global memory access, data input and program flow controls. The DPUOS is coded partly in DSP56001 assembly language and partly in C.

All high-level data processing algorithm and DPU task control is coded in C. Since the DPU board contains a limited amount of memory (only 32K words), it is not possible to fit all of the White DSP processing code onto the White DSP board. Thus, the White DSP code has been broken up into smaller units, called "swap units." The data processing is done by the DPUOS loading and executing individual swap units in a predetermined fashion.

3.2 Simulating the Overall DPU

The purpose of simulation is to construct a computer representation of the simulation target and execute the code as realistically as possible. However, there are limits. For example, it would require a tremendous amount of work to simulate the real-time, asynchronous, and unscheduled aspects of the system. Spending such a large amount of time is probably not justified at this point.

At the moment, the DPU simulation includes the following major characteristics:

1. The global RAM is simulated as a disk file called `global_ram` containing 5M words. On the DPU, each word is 24 bits. On a Sun/UNIX workstation, the default simulation environment, each word is 32 bits, i.e. an integer. The 32 bit words can simulate the 24 bits words without difficulty. Even though some

of the memory on the DPU will only have 16 bit dynamic range (small word memory), since they will appear as 24 bits words to the processors, we shall not make any distinction between the small and big word memory. The memory map of the `global_ram` file will exactly mimic the global memory on the DPU, with the exception that the DPU global memory addresses contain a gap between small word and big word memory, whereas this gap is currently not simulated. The global RAM will be accessible by all the DSP simulation codes.

2. The local on-board RAM on the White DSP is simulated in part by a file called `local_ram`. Since some of the local RAM will be occupied by DPUOS and some by the swap units, it is very complicated and probably impractical to simulate the local RAM in a full 32K word configuration. One the other hand, there are some numbers in the local RAM that contain critical parameters shared by swap units. Thus, all variables that are used by the swap unit code on the White DSP will need to be stored and passed on. For easy readability, the file `local_ram` is kept as an ASCII text file. In contrast, the `global_ram` is stored as a binary file due to file size considerations.

3. In the real XMMOM DEM, the DPU-ICU communication is done through the SSI interface on the White DSP and a specialized interface support on the ICU side. There is currently no real-time hand-shaking or low-level diagnostics such as DPU heart beats. The commanding by the ICU on the high-level DPU processing is simulated by a software manipulator called *icu_command* which modifies the `local_ram` and `global_ram` files, signaling a change of DPU status.

4. The access of global memory on the DPU is done on a round-robin, time-restricted honor system scheme. The real-time round-robin access is not simulated. However, the round-robin scheme constrains the amount of memory each DSP can access at a time. This factor influences implementation of the high level C codes of the processing algorithm. The simulation uses the same high level codes and interface to perform calculations, accepting the apparent inefficiency.

5. A separate file, `blue_local_ram`, is used to simulate critical areas of the Blue DSPs' local RAM. It consists mainly of arrays and tables necessary for the Blue DSPs' photon event and fast mode processing.

6. A separate file, `red_local_ram`, is used to simulate critical areas of the Red DSP's local RAM.

7. In real DPU operation the White DSP communicates to the Blue and Red DSPs through (1) the SCI interface for prompt commanding and (2) a common area in global memory ("PROC" memory) for non-time-critical bulk data. Simulation of the latter is straightforward. The former is simulated by an auxiliary C function *sci_command* which manipulates the `blue_local_ram` and `red_local_ram` files.

8. The Blue and Red DSPs communicate to the White DSP by writing to status words in specific locations of PROC memory. In the real DPU, some of these words, such as heartbeat, status, and error, are probed by the White DSP every millisecond, and special conditions trigger the White DSP to respond. This real-time aspect of White DSP probing is not simulated. Some other status words, such as the Red shift-and-add status word, are probed asynchronously by the high level C codes on the White DSP. Simulation of these activities will require careful coordination between the flight codes and the UNIX shell script that executes them sequentially.

9. A realistic data queue output is generated. It contains all data generated by the high level code presented to the ICU-DPU interface as valid DPU data blocks. These data can be converted into OM source packet format via a formatting facility.

3.3 Simulating the Blue DSP

The DPU requires two Blue DSPs to process the data from the MIC detector. The data are received from the MIC, alternately sent to each Blue DSP. The addressing calculation is performed and the image pixel on the global RAM is incremented. The actual Blue DSP processing on the DPU is highly optimized using assembly codes with hardware ping-pong between Blue DSP 1 and 2, buffering on the Blue DSP boards and the interleaving bus access. It is impractical to mimic the operations on the Blue DSPs exactly.

Instead, we concentrate on the address calculation aspect, as it is most important for downstream processing on the White DSP. It is straightforward to code the address calculation and memory incrementation in C. The memory location being incremented in the `global_ram` must be correct for the White DSP to extract that information and perform subsequent processing. This coordination is accomplished with the use of the address calculation parameters stored in global RAM accessible by all processors.

Current simulation code allows simulation of both image and fast modes. For fast mode operation, the simulation data must contain time markers since we cannot simulate the actual real time behavior. Insertion of time markers into the data is enabled by setting a compilation flag in the MIC simulator code (see below).

The Blue DSP simulation code receives configuration and status information from the White DSP code via the `blue_local_ram` file.

3.4 Simulating the White DSP

Currently, the high-level White DSP code consists of a main program and associated functions for traffic control, and the swap units which carry out specific tasks. The main objective of the White DSP simulation is to execute the code in a configuration as close as possible to the actual situation in the real DPU. The White DSP high-level code uses the same set of flight codes for simulation on a Sun/UNIX workstation and for execution on the DPU itself. There are a few small sections of "real" code that differ for the two modes. The Sun code has many additional sections that provide output for diagnostics. The differences are controlled by C preprocessor directives: A single flag (`#define` directive) in the White DSP code is used to distinguish versions when compiling. We have developed a set of utilities (UNIX shell scripts and make files) to make compilation relatively transparent to the user. Further, we have developed utilities to mimic the facilities provided by the DPUOS. See below for more discussion.

The `whitedsp` main program is a traffic control program that branches between different "tasks" or "states." The tasks are controlled either by the ICU or by DPU internal logic. The main program code is continuously resident on the White DSP board. The task processing involves the sequential execution of any number of swap units.

In developing the simulation scheme for the high-level White DSP functionality, we maintain the philosophy that we must use only one set of codes. However, there are facilities provided by the DPUOS which do not have direct correspondence in UNIX. In these cases, we have developed reasonable UNIX substitutes such that high-level codes that call those functions would remain the same and the interface with ICU would remain the same. The following are three major facilities for which we have developed UNIX substitutes:

1. The execution of the swap units on the DPU is done by passing the swap unit ID to a loader, which loads the code from global RAM (disk) onto the local RAM and executes. In simulation, the swap units are execute in the same manner by passing the swap unit ID to a program which simply calls the appropriate function. (Sun/UNIX workstations, the default simulation platform, usually have sufficient physical or virtual memory that there is no need to load individual codes from the disk.)

2. As discussed above, the memory access to the global RAM is done in a round-robin fashion among the four DSPs. The round robin bus access and timing aspects are not simulated. On the DPU, the global memory access is accomplished by invoking four DPUOS service functions. The UNIX substitutes for these memory access functions manipulate the global RAM area in the same manner as the DPU.

3. The communication between the White DSP and ICU involves real-time operation and is difficult to simulate. As described earlier, the input from ICU to the DPU is simulated by a manipulator that modifies the global and local RAM files, bypassing the DPUOS command processor. The output from the DPU to the ICU includes command passing and handshaking which is not being simulated. From the high-level algorithm's standpoint, communication and output of data are done by calling several functions serviced by the DPUOS. The UNIX substitutes of these functions simply write out the data

into a file called `queued_data`. The format of the data, however, follows exactly the interface definition between DPU and ICU. We have also implemented a telemetry processing routine that will properly packetize the DPU data blocks. The final outcome is a realistic telemetry stream that can then be processed by the analysis programs in the same way as the data generated by the real ICU.

3.5 Simulating the Red DSP

After the loss of the red detector, it was decided that the Red DSP should handle the shift-and-add calculations. As a result, the high level code for the Red DSP, called `reddsp`, is structurally very similar to the `whitedsp` code. After initialization is complete, `reddsp` runs in an idle state until commanded by the White DSP (via *sci_command* manipulation of the appropriate status word in PROC memory) to perform the shift-and-add. The shift-and-add code itself had been a swap unit executed by `whitedsp`, and is now a "task" within the `reddsp` code. When the shift-and-add is finished, its status word in PROC memory is updated to inform the White DSP that it may proceed to the next tracking frame when ready.

4 The Simulation Code

The $^{\mathrm{XMM}}$OM DPU simulation code consists of several individual programs and a number of other files required at run time. Each program performs a specific function as described below. The UNIX C shell scripts used to run the simulation programs are described in Section 6 and listed in the Appendix.

Currently, with the simplistic OPTICS and DETECTOR modules, we can generate 1.9 million events in about 3 minutes on SPARC 10/52, or about 10000 events per second for a full MIC frame with reduced resolution. The processing time is *not* reduced when the MIC is windowed. This is because photons falling outside of the windows are eliminated, i.e., do not generate a valid output, in the `mic` simulation module, very close to the end of the data chain. Although this sounds wasteful, it is more realistic. On *sdt1*, *sdt2*, and *ansel* ($^{\mathrm{XMM}}$OM Sun workstations at Penn State), each machine being a SPARC10 model 30, the processing time is about doubled. The `data.dump` file in the example above is large, consisting of 4 bytes per event. If one wishes to do a better job in simulating the quantum efficiency and the spectral response, then the processing time will be lengthened.

The real DPU and DPU simulation are concerned with processing the MIC events output by the real or simulated Blue DSP. Included in the term "processing" are tasks such as field acquisition, selection, identification, and tracking of guide stars. Before any of this processing can take place, there must be some initialization.

Due to memory restrictions of the real White DSP, all the high level codes for the White DSP cannot be resident simultaneously. Therefore, specific functions (e.g., window validation, data compression) are isolated as separate pieces of code which are compiled separately under DPUOS. These "swap units" are loaded by DPUOS only when required by the continuously resident portion of the code. We do not need to simulate this separation of sections of code in detail. Under UNIX, simulation versions of the continuously resident code (`whitedsp.c`) and the swap units (`su_dowindow.c, su_compress.c`, etc.) are compiled and linked into a single executable, called `whitedsp`.

Several of the simulation programs require input files whose contents are used to control the actions of the programs. These input files have `.pin` filename extensions to indicate that they are for *p*arameter *in*put. Some of the `.pin` files are created from templates at run time based on user specifications in the simulation scripts and simulation command line arguments (see below). Example `.pin` files are given in the relevant sections below. The line numbers and comments in the listings are not present in the distribution files, and should not appear in yours! Beyond that, the format must be followed exactly.

To summarize this section, we already have a basic structure and simple codes for the data chain running. DPU simulation uses the White DSP flight code and a simple ICU substitute. Results of simulations can be used to support lab testing.

4.1 The Data Chain Programs

4.1.1 Photons from the Field of View

The program `raygen` performs the tasks of the SCENE element of the data chain described above. Using a Monte Carlo technique, it generates rays corresponding to photons from sources in the field of view (FOV, see below). It can simulate point sources against a flat or non-flat background. Statistical fluctuations in source brightness and intrinsic variability may be present and are simulated. Output is one ray at a time. Its behavior is controlled by an input file called `raygen.pin`. The distribution version of `raygen.pin.template` looks like

```
sourcecatfile= Tsourcecat      ; the point source catalog filename
scenemapmaxx= 512              ; size of the scene map in x (pixels)
scenemapmaxy= 512              ; size of the scene map in y (pixels)
scenepixelsize= 2.2            ; pixel size of the scene map (arcsec)
bgdflux= 0.00015e0             ; background level (cts/sec/arcsec^2/cm^2)
sbgdfile= !                    ; scene map filename
bgdreplicate= 0                ; replication of scene map to cover FoV
tvsourcefile= !                ; time variable source filename
targettype= 0                  ; aperture type: 0 = circular, 1 = rectangular
targetdim1= 15.0               ; dimension of aperture (radius or primary mirror)
targetdim2= 4.61               ; dimension of aperture (radius of central obstruction)
driftfile= tdrift.dat          ; drift history filename
begintime= Tbegintime          ; begin time of photon generation (sec)
integratetime= Tintegratetime  ; total time of photon generation (sec)
randomseed= Trandomseed        ; seed for random number generator
logfile= !                     ; log file filename
```

The `gen_mic1` script (see below) edits in values for the strings beginning with `T`, and writes the result to `raygen.pin`. An exclamation point in a filename field indicates that the additional effect to be affected by that filename will be in the default configuration. For example, specifying `!` for the `sbgdfile` field means that the background will be treated as flat. The default configuration is usually fairly obvious, as in this example.

4.1.2 Pre-Telescope Radiative Transfer

The program `raytrans` is a template for incorporating radiative transfer effects between the sources and observatory, i.e., the TRANSFER element of the data chain. The current version simulates seeing with a Gaussian random deviates and a user-specified mean FWHM of the seeing disk. Output is one ray at a time.

Because OM is part of a space observatory, no atmospheric effects need to be considered. Indeed, the nature of the instruments is such that the TRANSFER element of the data chain probably can be neglected entirely. As of this writing, the simulation scripts do not run `raytrans`. Should you wish to use it, you will need a `raytrans.pin` input file like:

```
seeing_fwhm= 0.0               ; Gaussian FWHM of seeing (arcsec?)
randomseed= -18765             ; seed for random number generator
```

4.1.3 Telescope Motion

During an exposure, the spacecraft orientation with respect to targets in the field of view will change. The program `platform` represents the PLATFORM element of the data chain and provides the capability of simulating drift and roll of the spacecraft when calculating the position of a ray entering the telescope. Output is one ray at a time. The `platform` program requires `raygen.pin` as an input file. The program also requires a file, normally called `tdrift.dat` (see `raygen.pin`), that contains a description of the drift history for the exposure.

4.1.4 Telescope Optics

The program `tracesim` is a simple optics simulator that performs the tasks of the OPTICS element of the data chain. It takes a ray's directional vector, applies a Gaussian PSF to it and turns that into a position in the focal plane. This simulates a telescope focussed at infinity with a detector located in the image plane, the situation applicable to XMM-OM. Output is one ray at a time. Its behavior is controlled by an input file called `tracesim.pin`. An example of the input file is

```
focal_length= 390              ; effective focal length (cm)
optics_fwhm= 20.e-4            ; Gaussian FWHM of optics (cm)
randomseed= -18976             ; seed for random number generator
```

4.1.5 Detector

Currently there are two MIC simulator programs, `mic0` and `mic1`, which can perform the tasks of the DETECTOR data chain element. Each program takes a ray and, based on its position, turns it into a digitized position, reformats the photon into a MIC 24 bit event record if the photon is in a valid MIC window. `mic1` is a more elaborate MIC simulator than `mic0`. It also checks for local dead time: a photon will not be registered if it is incident on a pixel that has registered a photon less than a certain dead time prior. It also has a compilation flag to allow the insertion of MIC time markers into the data stream. Output from both programs is a group of MIC events at a time. This group corresponds to data collected within one commandable MIC CCD frame time. Program behavior is controlled by an input file, `mic0.pin` or `mic1.pin` as appropriate. The current simulation scripts invoke `mic1`, and edit a template version of the input file (`mic1.pin.template`) that looks like

```
pixelsize= 924.e-4             ; MIC pixel size (cm)
mic_binning= Tmic_binning      ; MIC binning factor
mic_xoffset= 0.9216            ; MIC origin with respect to the center
mic_yoffset= 0.9216            ;     of the FOV, x and y (cm)
mic_fwhm_x= 1.8e-3             ; MIC Gaussian FWHM in x (cm)
mic_fwhm_y= 1.8e-3             ; MIC Gaussian FWHM in y (cm)
dead_time= 0.001               ; pixel dead time (sec)
randomseed= -471               ; seed for random number generator
next_frame_time= Tnext_frame   ; frame counter
mic_frame_time= 0.01           ; MIC frame time (sec)
```

The `mic0` template input file `mic0.pin.template` is similar.

4.1.6 Electronics

Currently, two programs can be used to play the ELECTRONICS role in the data chain.

The program `bluedsp` is a straightforward Blue DSP simulator. Its basic function is to accept a MIC event, apply an address calculation and increment the corresponding memory location in the DPU global memory area (`global_ram` file). It also has a commandable blue fast mode with timing controlled by the embedded time marker in the MIC data stream (from `mic1`).

The program `bluedsp_eng` is a straightforward Blue DSP raw data mode simulator. This program will write 24 bit MIC events (without parity) to the global memory area for one million events. These events can later be dumped for diagnosis and testing.

Neither program simulates the two separate Blue DSPs. Input data for both is a group of MIC events from the DETECTOR data chain element. Input address calculation parameters are obtained from the `global_ram` file. There is no output to *stdout*. Instead, the events are registered by incrementing locations in the array that is eventually stored in the `global_ram` file.

4.1.7 Skipping Data Chain Elements

There is also a program called `datapass` that simply transfers input data to its output unchanged. `datapass` can be used in place of a intermediate data chain element program for cases in which we wish to neglect the effects of that element. Because `datapass` does not do anything to the data, it could be skipped altogether, depending on the details of the setup for running the simulations.

4.2 DPU Simulation Programs

4.2.1 Simulating the ICU

It is not possible to realistically simulate the interaction between the ICU and DPU. Nevertheless, we must have a means of approximating that communication that is satisfactory for simulation purposes. This simulation of ICU commands to the White DSP is accomplished with the program `icu_command`. It is a simple program that manipulates the RAM files read by the `whitedsp` program (see below).

The `icu_command` program requires one command line argument. This argument is a four-character hexadecimal code for the desired command. If a specific command requires arguments, they must follow on the `icu_command` command line. The hex command codes and argument requirements may be found in "XMM-OM ICU-DPU Protocol Definitions," project document XMM-OM/MSSL/SP/0014.

4.2.2 Simulating the White DSP

The program `whitedsp` handles most of the data processing duties. All the White DSP processing codes cannot be simultaneously resident on the White DSP board, due to its limited memory of only 32K words. Therefore, the White DSP flight code is designed so that sections of the code that perform specific tasks may be loaded into memory as needed. These sections of code are called *swap units*. In the Sun/UNIX simulation environment all the code can be present in a single executable without any significant loss of realism.

It should be noted that there are some differences between the versions of `whitedsp` for simulation and for use on the real DPU. C preprocessor directives in the code are used to tell the compilers which code to use for cases in which there are differences. Most of the code used for simulation is the actual flight code. Much of the non-flight code is present for log and debugging purposes. More importantly, there are a few critical differences in the control of code execution and progression dictated by the different natures of the simulation (UNIX) and real (DPU) environments.

In simulation mode, `whitedsp` is invoked without arguments. All execution controls of the high level flight codes are by the controlling parameters stored in the `local_ram` file. In this context, the `icu_command` code serves as the DPUOS command processor effecting control parameter changes that are visible to the high level flight codes.

4.3 Other Simulation Programs

There are several other programs used in a complete simulation run. They set up the necessary simulation environment and input parameters.

4.3.1 Sources in the Field of View

The first requirement for any simulation is a region of sky to be observed. Point sources can be simulated based on observed stellar statistics, or can be derived from actual star catalogs. The program `sourceinit` is responsible for generating a list of point sources in the field of view. Each source is specified by position in the field of view, apparent V magnitude, B–V color excess, and galactic coordinates. The V and B–V information is derived from the galaxy model of Bahcall & Soniera (1990, ApJS). The list constitutes the input for the `countrate` program. As of this writing, only simulated fields are supported. Source positions are calculated with a random number generator. The program creates a file called `sourcecat.dax`. Program behavior is controlled by an input file called `sourceinit.pin`. An example of the input file is

```
longitude= 20                ; galactic longitude (deg) for star density calc.
latitude= 20                 ; galactic latitude  (deg) for star density calc.
iseed= -3342                 ; seed for random number generator
exclude_left= 0              ; source exclusion zone left   boundary (radians)
exclude_right= 0             ; source exclusion zone right  boundary (radians)
exclude_bottom= 0            ; source exclusion zone bottom boundary (radians)
exclude_top= 0               ; source exclusion zone top    boundary (radians)
ext_sourcefile= !            ; filename for source catalog in exclusion zone
```

There are nine columns in the `sourcecat.dax` output file. Because you may want to generate your own source file for input to `countrate`, or edit the one created by `sourceinit`, we describe the contents here. The items below correspond to columns in the file (in order).

1. Source x location (arcsec). It is measured with respect to the optical axis, i.e., center of the detector.
2. Source y location (arcsec).
3. Source apparent V magnitude (mag).
4. Source B–V color excess (mag).
5. Source spectral type (0-6 corresponds to OBAFGKM).
6. Source spectral subtype (only [0258] are allowed, and not all of those for every type and luminosity class – see `countrate`).
7. Source luminosity class (0-6 correspond to Ia, Ib, II, III, IV, V, VI, and VII (white dwarfs), respectively).
8. Field of view galactic longitude (deg). This should be the same for each entry in the file.
9. Field of view galactic latitude (deg). This should be the same for each entry in the file.

   However, in `sourcecat.dot`,

As of this writing work is in progress to allow real regions of the sky to be considered, e.g., by reading the HST Guide Star Catalog. This function will not be part of `sourceinit`. The code is being written as part of the Science Data Terminal (SDT) software, and is therefore an alternative to `sourceinit` for generating the `sourcecat.dax` file. See The Science Data Terminal User's Guide ($^{\mathrm{XMM}}$OM document XMM-OM/PENN/ML/0003) for details and instructions on use of that software. Scott Horner (horner@astro,psu.edu, 814-863-6038) is the contact person for the SDT software. Note that the Guide Star Catalog is incomplete at magnitudes of interest.

4.3.2 Defining Time-Variable Sources

The program `tvsourceinit` is used to define the characteristics of time variable sources in the field of view. (`sourceinit` generates only sources of constant brightness.) The program creates a file called `tvsourcecat.dat`. The current simulation scripts do not run `tvsourceinit`, i.e., current default simulation does not include intrinsically variable sources.

4.3.3 Calculating Source Count Rates

The $^{\mathrm{XMM}}$OM Science Simulator program (see document XMM-OM/MSSL/ML/0002.01 by Phil Smith) has been integrated into the OM data simulation scheme. This version is known as `countrate` because it takes the magnitude, color excess, and spectral type information from `sourcecat.dax` and calculates count rates for each source.

The program uses several input data files. Those files contain information about stellar energy distributions and wavelength-dependent efficiency of various optical elements. Thus, the `sourceinit-countrate` combination produces countrates for realistic observing situations: output is count rates for realistic distributions of stars at any galactic coordinates observed with any of OM's filters.

`countrate` also calculates the background (dominated by diffuse galactic and zodiacal light) count rate. It currently does not support date specification, assumes a 90° sun angle.

The output from `countrate` is written to a file called `sourcecat.dat`. This file is later renamed by the `sexpose` (or `sexposeng`) script for input to `raygen`. The `sourcecat.dat` files has the following columns:

1. Source type (integer). For point sources, specify 1. For background, specify 0. In the current version of `raygen` there is no way to treat extended sources other than the overall background. The background map is *not* specified here. It is specified in `raygen.pin`.
2. Source x location (arcsec). It is measured with respect to the optical axis, i.e., center of the detector.
3. Source y location (arcsec).
4. Source apparent V magnitude (mag).
5. Source count rate ($\mathrm{cts\,s^{-1}\,cm^{-2}}$). The `raygen` program generates another file called `sourcecat.dot` in which the corresponding column has the aperture size folded in, i.e., data are in $\mathrm{cts\,s^{-1}}$ in that file.
6. Field of view galactic latitude (deg). However, in `sourcecat.dot`, generated by `raygen` for debugging purposes, data in the corresponding column represent the accumulated probability (from 0 to 1) that a photon is due to a source up to the current line of the file. Those data are used in `raygen` to determine which source is responsible for each photon, through a Monte Carlo technique.
7. Three columns that are not currently used by `raygen`. `countrate` fills them with zeros. Althought these columns are not used, they must be present.

4.3.4 Generating a List of Reference Stars

The program `gen_ref_stars` reads the `sourcecat.dat` file and up to 16 brighter stars from the input listing in `sourcecat.dat` as reference stars for field acquisition. Output is written to a configuration file called `ref_stars.cfg` which will be written into simulation memory by `icu_command`. Input to the program is via the `mic1.pin` and `tracesim.pin` input files. To override the random number generator seeds in those files, `gen_ref_stars` has its own input file, called `gen_ref_stars.pin`, which contains just one line:

```
randomseed= -515              ; seed for random number generator
```

4.3.5 Generating a Spacecraft Drift History File

As stated above, `platform` can incorporate variable orientation of the spacecraft with respect to targets in the field of view into its calculations of source coordinates. To do this, it requires information on the spacecraft motion. The program `mkdrift` generates a file containing a prescribed drift history for the spacecraft. The file is called `tdrift.dat`. Program behavior is controlled by an input file called `mkdrift.pin`. An example of the input file is

```
iseed= -123                   ; seed for random number generator
dxl= 0.2e-9                   ; long term drift in x per dtdrift (rad)
dyl= 0.2e-9                   ; long term drift in y per dtdrift (rad)
drs= 0.25e-6                  ; random walk step size per dtdrift (rad)
dth= 0.25e-6                  ; linear roll per dtdrift (rad)
dlimit= 2.5e-5                ; limit of the random walk (rad)
dthmax= 2.5e-5                ; maximum roll angle (rad)
dtdrift= 0.1                  ; drift time step (sec)
begindrifttime= 0.0           ; begin time of drift calculation (sec)
totaldrifttime= 1100.0        ; total time of drift calculation (sec)
velocity= 0.0                 ; not used
tha0= 0.0                     ; not used
phia0= 0.0                    ; not used
alt= 0.00                     ; not used
```

The format of the `tdrift.dat` file is as follows:

Line 1: ndtdrift dtdrift begindrifttime
Lines 2 – ndtdrift+1: six column on each line as follows:

Column 1: drift in x (yaw), in radians
Column 2: drift in y (pitch), in radians
Column 3: roll (roll), in radians
Columns 4 – 6: not used for XMM-OM simulation.

4.3.6 Formatting Data for Telemetry

The program `tm_packet` is used to format the final observatory data into valid telemetry packets for delivery of data to eager astronomers on the ground.

4.4 UNIX Simulation C Shell Scripts

An entire simulation run can be started with a single command that executes a UNIX shell script. The shell script is a file that contains valid UNIX commands. Those commands may invoke UNIX commands, user programs, or other scripts. The main DPU simulation script is called `simom`, for **sim**ulate OM. Other scripts currently used in DPU simulation are `gen_mic1`, `sexpose`, `sexposeng`, `sim_clean_slate`, and `timed_message`. Commands in the scripts are invoked in a sequence that duplicates real $^{\mathrm{XMM}}$OM operation as closely as possible. These scripts will be discussed in detail in Section 6.

4.5 Summary

In summary, we have developed a working suite of programs and controlling scripts to facilitate $^{\mathrm{XMM}}$OM simulations. The scheme's features include:

- Modularity. This facilitates the ability to vary the degree of realism when running simulations, and the ability to interchange between software modules and digital hardware.
- Duality. This allows compilation and execution of the same software modules in the simulation environment and on the real DPU hardware.
- Adherence to data interface definition at all stages of simulation. This allows one to develop and exercise the analysis environment with the simulated data.

5 Installing the Simulation Package

This section provides instructions for obtaining private copies official $^{\mathrm{XMM}}$OM project code and associated files. Following that are instructions for installing the DPU simulation software. All those instructions assume that you have direct access to Penn State Astronomy & Astrophysics Department Sun workstations. If you do not, contact Scott Horner (horner@astro.psu.edu) or Alan Welty (welty@astro.psu.edu) for directions for getting the necessary files.

There are some differences between The White DSP Sun Simulation code and that used on the DPU itself. There are also somewhat different steps needed to build the executable files. Because most use of the simulation code outside the core software development team will be performed entirely on Sun workstations (or another computer platform) rather than on the real DPU, this section will concentrate on that mode of use.

5.1 UNIX Environment

Before you attempt to install or run any ^{XMM}OM code, you should be sure to set up an appropriate UNIX environment. Many of the utilities and programs within the ^{XMM}OM hierarchy make significant use of UNIX environment variables and command aliases. Because the executable ^{XMM}OM programs and shell scripts will be placed in your personal directory tree, rather than in a standard system location, your `path` (list of directories searched for commands) must also be modified. At Penn State, there is a primary ^{XMM}OM computer account on a Sun workstation known as *ansel*. Among other things, it contains useful files for creating a suitable UNIX environment. The author achieves this by executing the following UNIX commands at login time (in his `.login` file):

% `setenv MYXMMOM /bulk/ramsey3/xmmom` # Substitute an appropriate personal path.
% `source /home/ansel/xmmom/xmmom.cshrc`
% `source /home/ansel/xmmom/xmmom.alias`

Note that "%" is used here as a generic UNIX prompt, not part of the command. Only the characters in `this font` should be typed after your UNIX prompt or entered into your `.login` file.

At a non-Penn State site, some editing of the two files *source*d above may be required. The first, `xmmom.cshrc`, is the important one. If you have an account on *ansel*, you can FTP them:

% `ftp ansel.astro.psu.edu`
[Enter username and password when requested.]
ftp> `cd /home/ansel/xmmom`
ftp> `get xmmom.cshrc`
ftp> `get xmmom.alias`
ftp> `quit`

Edit appropriately modified `setenv` and `source` commands, like the three above, into your `.login` file. To get the environment set up for your current login session, either enter those commands individually, or type

% `source .login`

To verify that the correct environment has been established, type

% `setenv | grep XMMOM`

In particular, check to see that the path for `XMMOMSIMHOME` is set correctly. It should be equivalent to `$MYXMMOM/sim`. In the author's case the substitution for `$MYXMMOM` yields `/bulk/ramsey3/xmmom/sim`.

5.2 Obtaining the Necessary Files

Before you can install the suite of simulation programs and associated files you must locate an official copy of the files and copy them into your `$MYXMMOM` directory tree. (Note that the "$" character must be prepended to the environment variable's name to achieve the desired string substitution.) The official repository for ^{XMM}OM files resides in the xmmom account on *ansel*. The necessary files may be found under its `/home/ansel/xmmom/cvs/master` directory, but should not be copied directly. Members of the UNIX group "xmmom" on *ansel* may use the `cvs` utility (a convenient front end to RCS, the "revision control system") to get copies of the files. The following commands will suffice:

% `cd $MYXMMOM`
% `xmmom_checkout`

The `cvs` commands in the `xmmom_checkout` script will generate a number of messages on your terminal.

Some of the ^{XMM}OM code is also available in UNIX tar files. You can obtain and unpack such files as follows:

% `cd $MYXMMOM`
% `ftp ansel.astro.psu.edu`
[Enter username and password when requested.]
ftp> `bin`

```
ftp> cd /home/ansel/xmmom
ftp> get whitedsp.tar
ftp> get simom.tar
ftp> quit
% cd $XMMOMWDSPHOME
% tar xvf $MYXMMOM/whitedsp.tar
% cd $XMMOMSIMHOME
% tar xvf $MYXMMOM/simom.tar
% rm whitedsp.tar simom.tar
```

If the desired tar files do not exist, please contact Scott Horner or Alan Welty at the addresses given above.

5.3 Building Simulation Code for Sun

Once the files have been copied into your directory tree, you must compile and link the programs, and place all the executables (programs and shell scripts) into a single directory specifically for them. There are two main steps in this process. The first is to build a library that will be linked with some of the simulation code. The following commands should accomplish that:

```
% cd $MYXMMOM/sim
% mkdir lib
% cd src/aparm
% gmake empty install clean
```

This assumes that the Gnu make utility (`gmake`) is installed on your system. If not, `make` should also work in this case. A UNIX library file for the `aparm` package will appear as `$MYXMMOM/sim/lib/libaparm.a`.

The other main step is to compile and link the White DSP code and the suite of simulation programs, and to copy the executable files into the proper directory. As discussed earlier, C preprocessor directives are used to control which sections of White DSP code are compiled and linked for Sun simulations, and which are compiled and linked for DPU simulations (i.e., the actual flight code). In the current scheme, the `#define SUN` directive resides in `$MYXMMOM/dpu/development/include/setup.h` and is commented. That is, the code is configured for the DPU flight version by default. We have written a shell script and makefile system that automatically uncomments that statement, builds the Sun simulation version of the code, moves files to their correct destinations, and restores the original `setup.h` file. The following commands should complete the installation:

```
% cd $MYXMMOM/sim/src
% make_all new
```

The `make_all` script uses the Gnu C compiler, `gcc`, as well as the `gmake` utility. If `gcc` is not installed on your system, you can try editing the script to use a different C compiler. Edit the line "`CC = gcc`" to read "`CC = cc`" and try running `make_all new` again. If the compiler complains, your local `cc` compiler probably is not ANSI compatible. In any case, there will be a few warnings about `min` and `max` macro and `DEBUG` compiler flag redefinitions. They are expected and may be ignored. When the `make_all` script has finished, the executable programs and shell scripts can be found in the directory `$MYXMMOM/sim/bin`, which should be in your UNIX path (see Section 5.1). Enter the command

```
% ls -lF $XMMOMSIMHOME/bin
```

to verify that they exist and have the correct permissions. You should see at least the following programs and scripts: `bluedsp`, `bluedsp_eng`, `countrate`, `gen_mic1`, `gen_ref_stars`, `icu_command`, `mic1`, `mkdrift`, `platform`, `raygen`, `raytrans`, `reddsp`, `sexpose`, `sexposeng`, `sim_clean_slate`, `simom`, `sourceinit`, `timed_message`, `tm_packet`, `tracesim`, `tvsourceinit`, `update_dat`, and `whitedsp`. If any of these are missing, go over the instructions again to be sure that all steps have been completed properly. If they have, then consult a simulation software maintenance person for help.

5.4 Building Simulation Code for DPU

We can run the flight codes on the actual hardware. However, we need input data. Until the observatory is in space obtaining real data, test data must be produced by the data chain simulation programs and collected in a file. Therefore, you must follow the directions in the previous subsection to build the simulation programs.

Compilation and linkage of the DPU flight code are accomplished by executing three shell scripts written for that purpose. The code is configured for the flight version by default. Once you have set up your UNIX environment for $^{\text{XMM}}$OM-related work as described above, the following commands build the flight codes for execution on actual hardware:

```
% makebdsp
% makewdsp
% makerdsp
```

6 Running Simulations

As with building and installing the simulation code, there are differences between Sun and DPU when running simulations. Currently, The Sun simulations are run at UNIX level. They make heavy use of shell scripts to make running realistic simulations as easy as possible. Running the code on the DPU requires some preliminary work on the workstation with simulation programs and uses the VX Works software to control execution on the DPU.

6.1 Setting Up a Simulation Directory Tree

Whether you will be running DPU simulations on a Sun workstation (or other computer platform) or on the DPU itself, you will need to run the data chain programs to create data. Before attempting to run any of the $^{\text{XMM}}$OM simulation software, you must take care of a few details. The first is to have your UNIX environment set up properly. See Section 5.1 for details. The next step is to create a directory in which you will run the simulations. It should be separate from the simulation code directory tree. The author chose to build his simulation directory on *ansel* as follows:

```
% cd /bulk/ansel7/welty                    # change to a directory that already exists
% mkdir SIM
```

In the example commands below, this directory will be called `$mysim`. Make the appropriate substitution for your case. This directory should reside on a UNIX file system that has a large amount of free space. Data for simulation of a typical $^{\text{XMM}}$OM exposure of 1000 sec (100 ten second frames) requires up to 800 MB of disk space! A complete simulation run should include more than one exposure. The `global_ram` file requires an additional 21 MB per simulation run.

The current implementation of the simulation control scripts requires that your main simulation run directory have a subdirectory for each exposure included in your simulations. Further, the names of those subdirectories must be the same as the six character hexadecimal code for the corresponding exposure. As of this writing there is no official protocol for choosing exposure IDs, but we have been using a system. Please do not invent arbitrary exposure IDs out of thin air. If any of your OM simulation activities will be used in hardware testing or compared with official benchmark simulations or others, it will be important to follow a uniform exposure ID selection protocol. At of this writing, Howard Huckle (heh@mssl.ucl.ac.uk) is the person in charge of assigning exposure IDs. Cheng Ho should also be consulted before choosing exposure IDs for any simulations that will not be "for your eyes only." For the instructions below we will use the following exposure codes/directory names: `f80001`, `f80002`, etc.

Each exposure subdirectory must be set up to contain several input files that will be used to control the simulated exposure (see Section 4). This can be accomplished as follows:

```
% cd $mysim
```

```
% mkdir f80001 f80002 f80003 # etc.
% cd f80001
% cp -p $MYXMMSIMHOME/scratch/* .
% cd ../f80002
% cp -p $MYXMMSIMHOME/scratch/* .
etc.
```

In each of the exposure subdirectories there will be several `.pin` and `.pin.template` files. These are templates of parameter input files required by several of the simulation programs.

If any of the required `.pin` or `.pin.template` files are not present, they must be created in the exposure directories before you start a simulation run. Edit these files as desired to control the details of each simulated exposure. Examples of the parameter input files are given in Section 4. Be sure to follow the format of each file exactly.

There should also be files called `scw.cfg` and `mmw.cfg`, which define the user specified science and memory window configurations. Edit the `.cfg` files to specify windows as desired. The first line of each file contains a single number, the number of windows to be specified. This may range from 0 to 16. (The DPU observation operation allows up to 16 science windows. Consult the observation mode requirement document for further details.) There must be that many more lines to specify the windows. The following is an example `scw.cfg` file (as before, ignore the line numbers).

```
2
1032 1032 496 496 0 1 1 0 0 0 0
1056  528  32  32 0 0 0 0 0 0 1
```

where the first line gives the number of science window specifications to follow, and subsequent lines specify those windows into the detector. The science window specifications are given as follows: The first two columns specify the x and y values of the bottom-left corner. The next two columns specify the x and y sizes of the window. The fifth column specifies the observing mode: values of 0 and 1 indicate image mode and fast mode, respectively. (In addition, a mode value of 7 returned by the on-board window/memory management algorithm indicates a tracking guide star window. This mode cannot be requested by the user.) Columns 6 through 9 specify window parameters. See relevant documentation for science window parameter specifications. The tenth column specifies the number of the memory window to which each science window should be assigned. Window numbering begins with zero. There must be a one-to-one correspondence between user-specified science and memory windows. This user identification of science windows with memory windows is ignored in the current version of the flight code, which makes these assignments itself during window validation.

The following is an example `mmw.cfg` file.

```
2
1024 1024 512 512 0
1040  512  64  64 0
```

where the lines are used as in `scw.cfg`. In the window specifications, the first four columns have the same meaning as in `scw.cfg`. The fifth column specifies a window base address, which is assigned by the DPU flight code and may not be specified by the user.

Science Data Terminal (SDT) software under development as of this writing will allow interactive specification of windows on the real sky (as given in the Hubble Space Telescope Guide Star Catalog) and creation of corresponding `scw.cfg` and `mmw.cfg` files. Consult the SDT User's Guide (Document XMM-OM/PENN/ML/0003) for details.

There may also be source catalog and drift history files (`sourcecat.dax` or `sourcecat.dat` and `tdrift.dat`). The simulation script will create these files if they are needed and do not exist by running `sourceinit` and `mkdrift` as appropriate. You may run those programs yourself prior to a simulation run if you want to. Within the SDT software there will also be the capability of generating a source catalog file (`sourcecat.dat`) corresponding to real regions of the sky using the Guide Star Catalog.

6.2 Running Simulations on a Sun

Once the simulation programs and scripts have been installed and your personal simulation directory has been set up, you will be ready to run simulations. Before doing so, please read this entire section.

The top-level simulation script is called `simom`. The `simom` script invokes other simulation scripts and programs in a sequence that matches actual $^{\mathrm{XMM}}$OM operation as closely as possible. The `simom` script accepts a few option flags that will be discussed below. Samples of `simom` and other relevant scripts are given in Appendix A.

Take some time to get familiar with the scripts and how long they take to run. Listings of the scripts, including basic comments, are given in the Appendix. On *ansel*, a complete one exposure (1000 sec) run takes a few hours to generate the 100 MIC frames needed. The remaining tasks add about 40 minutes for about 12 minutes of CPU time. The bottleneck is I/O: if your disk makes noise when accessed, you'll hear it almost constantly as the `global_ram` file is manipulated.

6.2.1 Basic Usage

The essential command syntax for a single exposure simulation is

```
% cd $mysim                      # Be sure you're in the proper directory.
% simom f80001
```

If you want to continue working in the window in which you invoke `simom`, you can run the simulation in background by appending the `&` character to the `simom` command line:

```
% simom f80001 &
```

As noted above, OM simulations consume system resources. Be nice to yourself and others with whom you may share the computer by running simulations at low priority. This way, interactive use of the workstation will not be affected significantly and the simulation will have access to otherwise available resources on demand. This is accomplished with a command like

```
% nice +10 simom f80001
```

If you run the Bourne shell rather than the C shell, then specify `-10` rather than `+10` as the first argument to `nice`.

You can also submit the simulation run as a batch job as follows:

```
% at 12:00 simom f80001
```
or
```
% batch simom f80001
```

for SunOS 4.1.3 systems, or

```
% echo 'csh simom f80001' | at 12:00
```
or
```
% echo 'csh simom f80001' | batch
```

on systems running Solaris.

6.2.2 Options and Arguments

There may be occasions when you want to repeat a simulation run for one or more exposures. Changes in the simulation code constitute a good reason for rerunning a simulation. Data for the acquisition and reference frames and all the frames that contribute to the total exposure are saved by default. Those data may be reused. To tell `simom` you want to reuse existing data, invoke it with a `-t` flag, e.g., `simom -t f80001`. Reusing data saves a lot of computer time. If you forget the `-t` flag and do not modify relevant inputs, the new data will be identical to the old, and much computer time (and real time) will have been wasted.

In practice, every ^{XMM}OM *pointing* will require an acquisition frame that the White DSP flight code will use to "acquire" the desired field. The current incarnation of `simom` does not assume that an acquisition frame is required for each *exposure*. To instruct `simom` to perform field acquisition, invoke it with an `-a` flag.

By default, `simom` goes about its business pretty silently. If you want it to provide you with more information about what it is doing and how it is progressing, use a `-v` flag.

By default, `simom` sets up an exposure with 100 tracking frames. To specify a different number of frames, use the `-n` flag followed immediately, i.e., without whitespace, by the desired number of frames, e.g., `-n30`.

By default, `simom` sets up an exposure with the V filter. To specify a different filter, use the `-f` flag followed immediately by the filter code number, e.g., `-f7`. The filter IDs as understood by `simom` are: 0 for the blocked position; 1 for the "clear" position; 2 for UVW2; 3 for UVW1; 4 for U; 5 for B; 6 for V; 7 for UVM2; 8 for the magnifier; 9 for grism 1; and 10 for grism 2. Note that this flag applies only when the `-t` flag is *not* in effect, i.e., when new data are to be generated. If both the `-f` and `-t` flags are specified, the `-f` will be ignored silently.

By default, `simom` sets up for a normal science observation. To run an engineering mode simulation, use the `-e` flag followed immediately by the engineering mode code, e.g., `-e6`. The engineering mode codes are: 0 for raw data mode with both processors; 1 for raw mode with Blue DSP 1 only; 2 for raw mode with Blue DSP 2 only; 3 for centroiding data; 4 for a high resolution full frame; 5 for centroidind data confirmation; 6 for intensifier characteristics.

The other option flag `simom` recognizes is `-h`. This flag causes `simom` to print a message that describes correct usage and recognized arguments. After the message is printed, `simom` exits. Thus, the `-h` flag cannot be used when you want to run a simulation.

The `simom` script is written so that option flags and exposure identification arguments may be given in any order. The option flags may *not* be combined into a single flag, i.e., arguments like `-av` will not be recognized.

An argument that is not recognized as a valid option flag is interpreted as an exposure identification string. If any such argument is invalid, i.e., is not a valid exposure ID string with a corresponding subdirectory, the simulation run will ignore it.

In the current scheme, a new three exposure simulation run could be started with a command like

```
% nice +10 simom -a f80001 f80002 f80003 &
```

If you want to test changes in the `bluedsp` program or the simulation versions of the White or Red software, you would rerun the simulation later with the command

```
% nice +10 simom -a -t f80001 f80002 f80003 &
```

so that the data generated in the initial run are reused (i.e., this example is not for testing changes in data chain programs). In both commands above, the field acquisition step will be performed.

6.2.3 Further Control

Before running a serious simulation you probably will want to take some time to modify some of the scripts and/or input files so that each observation in a run is not identical! Whether you are exercising the software or attempting to simulate real circumstances, some input file modification is in order. One of the purposes of running simulations is to verify that the flight code does what it is supposed to in a wide variety of circumstances.

The programs that make needed data files (e.g., `sourceinit` and `mkdrift`) and the data chain programs require one or more input parameter files (`.pin` files) as discussed and listed in Section 4. Modify them to control generation of data as desired.

To inform `simom` that a simulation exposure is not the first in a pointing and that no acquisition frame is desired, you must do this in the appropriate exposure directory:

% `touch same_pointing`

This creates an empty file called `same_pointing`. If `simom` sees that file and it is not the first exposure of the simulation run, it will not tell `sexpose` to obtain an acquisition frame. There is no harm in obtaining an acquisition frame for each exposure in a multiple exposure pointing.

To change the characteristics of the individual exposures, you will need to see how `simom` works. The `simom` script takes care of the basic organizational aspects of a simulation run, and invokes `sexpose` to carry out most of the actual simulation tasks. It does not invoke `sexpose` directly, however. If, for example, the current exposure ID is `f80001`, then `simom` will look for another script called `simom.f80001`. If it exists, `simom` will execute it. If it does not exist, `simom` will create it and then execute it. The `simom.f80001` script should contain a single command: an invocation of `sexpose` with an appropriate set of arguments. Examine the `simom` and `sexpose` listings in the Appendix to see how this works.

The arguments `sexpose` expects to get from `simom.`*exposure_id* are

1. The name of the directory in which the simulation RAM files are located. Currently, this is set as the first exposure directory, but can be set to the main simulation directory.
2. The exposure ID string. Again, this must be a 6 character hexadecimal number. To ensure unique exposure IDs for simulation runs for future reference, please consult the published procedure for exposure ID assignment. Do not invent arbitrary exposure IDs out of thin air.
3. The number of frames that will make up the exposure. Normally, this argument will be used to control the total exposure time (see next argument).
4. The frame time (seconds per frame). This is the integration time per MIC frame, normally 10 seconds. Total exposure time is then the product of this and the previous argument.
5. The initial frame number. This will be -1 when an acquisition frame is required (the normal case), and 0 when the acquisition frame is not required, in which case a reference frame will be the first to be obtained. (Even in cases of multiple exposure pointings, a reference frame will be obtained for every exposure. The multiple exposures will typically be taken with different filters, and there is no reason to expect the set of guide stars chosen would necessarily be the same.)
6. A random number generator seed. This seed replaces the string `Trandomseed` in `raygen.pin.template` for output into `raygen.pin`. `raygen` is run only when data are being generated (i.e., no `-t` flag). The seed is incremented in `simom` (by 67, currently) for each frame, so that `raygen` doesn't produce identical output for each frame. (This is how we simulate statistical fluctuations in source brightness.)
7. A "dump frame" ID for debugging purposes. Leave this value at -1000 under normal circumstances. If frame dumps are needed, the user is advised to make a specialized copy of the `sexpose` script in the target exposure directory. Explicit modifications to that script will be required. Contact the DPU software development team for further details.
8. The through flag. If this number is 1, `sexpose` will expect to find pre-existing frame data files. If any of the required files do not exist, the simulation run will continue, but will be worthless because frame data will not have been placed in memory as needed. Any other value for this flag will cause `sexpose` to invoke `gen_mic1` to generate the necessary data by running the data chain programs.
9. The name of the actual `global_ram` file. Currently, the actual file is created in the `/tmp` directory to enhance performance (taking advantage of SunOS's "marriage" of the `/tmp` directory and system swap space). To allow more than one simulation to be run at the same time, the file is named with the time the simulation was started. For example, starting a simulation run at 3:05 pm produces a global RAM file called `150500_ram`.
10. The filter ID number (0–10; see §6.2.2).
11. The exposure mode (-1 – 6; see §6.2.2).

An example of when you might want to create a `simom.`*exposure_id* script before running a simulation might be when you want to change the total exposure time. The default when `simom` creates the script is 100 ten second frames, for 1000 seconds. To change the exposure time to, say, 500 seconds, the preferred method is to make a `simom.`*exposure_id* script that specifies 50 ten second frames.

Another example might be in the case of multiple exposure pointings. For all but the first, you could create `same_pointing` files to prevent `sexpose` from obtaining acquisition frames. You may also use the `simom.`*exposure_id* script to control this by specifying 0 is the initial frame, rather than -1. Indeed, if a `simom.`*exposure_id* file already exists in the exposure directory, you must edit in the desired initial frame number because `simom` will execute that script and ignore a `same_pointing` file.

Note that the simulation scripts and programs expect certain files to be present. On a first run of a simulated exposure (not a "through" run), some files will be created during the simulation. Do not delete any of those files (except possibly the `*.pin?` files, where `?` is a wildcard for a single character); they will be needed for any subsequent "through" runs.

The Appendix gives examples of the major simulation scripts with which you should become familiar before running simulations. In general only data chain program input files (`.pin` files, which are listed in Section 4) and science and memory window configuration files (`.cfg` files) should be modified. The scripts are already set up to run things in a way that mimics the actual OM situation. *If you do modify the scripts, be certain you know what you're doing!* One "legitimate" change you might make to a script is to break the data chain pipe and redirect data to a file as discussed earlier.

A C Shell Script Listings

This appendix contains listings of the UNIX C shell scripts that are used to run the simulations. By studying them you will be able to see the sequence of events in real $^{\mathrm{XMM}}$OM observations. Places where you are most likely to edit the files are indicated in the comments. In-script comments begin with the `#` character.

The following script, `timed_message` is used only to put timestamped messages in the simulation logfile. It is used frequently in the major simulation scripts which follow. The leading line numbers in the listings below are not part of the files.

```
#! /bin/csh -f

echo `date`" -> "$1

```

The top level simulation script, `simom`, handles the operational tasks that are executed once per simulation, and executes the `sexpose` script within a loop over the specified exposures. `sexpose` runs the tasks necessary for each exposure. `simom` executes another script, `sim_clean_slate`, to take care of some of the initialization. `sexpose` executes the script `gen_mic1` to run the data chain programs. It is in `gen_mic1` where you may break the data chain to redirect output to a file for examination. `simom`, `sim_clean_slate`, `sexpose`, `sexposeng`, and `gen_mic1` are shown below.

A.1 simom

```
#! /bin/csh
#+ ------------------------------------------------------------------------------
#
# NAME:
#      simom
#
# PURPOSE:
#      This script is the top level XMM-OM simulation script.
#      Most of the actual simulation is done by the "sexpose" script
#      which is invoked herein.
#
# INPUTS:
#      none
#
# CALLS:
#      sexpose         (via simom.$exp created and executed herein)
#      sim_clean_slate
#
# CALLED BY:
#
# RESTRICTIONS:
#      The environment variable XMMOMSIMHOME must be properly defined.
#
# MODIFICATION HISTORY:
#      07Aug95   ADW   add comment header, update other comments
#
# RCS $Id: simom,v 1.13 1996/05/10 22:41:35 welty Exp $
#
#- ------------------------------------------------------------------------------

unset noclobber
umask 002
unalias rm
unalias mv
unalias cp

# --- Define the symbolic command names

$XMMOMSIMHOME/src/idiot_convert > idiot.csh
source idiot.csh

#

# --- Set default control parameters.

set iframe = 0      # default first frame number; 0 => REF frame
set filter = 6      # default filter ID
set eng_mode = -1   # default engineering mode. -1 means not engineering mode
set frames = 100    # default frames/exposure
set thru = 0        # 0 => make new data; 1 => use existing data
set verbosity = 0   # 0 => normal; 1 => verbose
set exp_seq = ( )

# --- Process the command-line arguments.

foreach arg ( $argv )
   switch ( $arg )
      case '-h':
         echo ' '
         echo 'simom is a C-shell script to run simulated exposures for XMMOM.'
         echo ' '
         echo 'Usage:'
         echo '   simom -h'
         echo '   simom [-a] [-f#] [-n#] [-t] [-v] exp_id'
         echo ' '
         echo 'Options: -h: print this message'
         echo ' '
         echo '         -a: start with acquisition frame'
         echo '         -t: expose with existing data'
         echo '         -v: turn on verbose output'
         echo ' '
         echo '         -e#: engineering mode ID (# in 0-6;  default = -1)'
         echo '         -f#: specify a filter ID (# in 0-10; default = 6 [V])'
         echo '         -n#: specify # frames (default = 100)'
```

```
         echo ' '
         echo 'exp_id:  one or more 24 bit exposure id number(s) in hex.'
         echo '         Exposures reside in directories with the same names.'
         echo '         The directory must be set up by the user beforehand.'
         echo ' '
         exit
      case '-a':
         set iframe = -1
         breaksw
      case '-e[0-6]':
         set eng_mode=`echo $arg | sed -e 's/-e//'`
         set iframe=-1
         set frames=0
         breaksw
      case '-f[0-9]*':
         set filter=`echo $arg | sed -e 's/-f//'`
         breaksw
      case '-n[0-9]*':
         set frames=`echo $arg | sed -e 's/-n//'`
         breaksw
      case '-t':
         set thru = 1
         breaksw
      case '-v':
         set verbosity = 1
         breaksw
      default:
         set exp_seq = ( $exp_seq $arg )
         breaksw
   endsw
end

#

# --- Put exposure IDs into a character array and find the number of exposures.

set exposures = ( $exp_seq )
set n_exposures = $#exposures

if ( $verbosity == 1 ) then
   echo $1 $2
endif

echo 'simom:' $thru $iframe $frames $eng_mode $exposures

# --- Exit if there are no exposure IDs given.

if ( $n_exposures < 1 ) exit

# --- Set up directory specifications.

set exp = $exposures[1]
set parent_dir = `pwd`

cd $exp                 # comment if ram files should go in parent_dir
set working_dir = `pwd`
cd $parent_dir

# --- Create RAM files, fake loading DPUOS, and initialize.

set global_ram_file = `date +%H%M%S`_ram

cd $working_dir
$XMMOMSIMHOME/bin/sim_clean_slate $global_ram_file
cd $parent_dir

#

# --- Loop over the specified exposures.

set i = 1
while ( $i <= $n_exposures )

   # --- Attempt to do the work only if the exposure directory exists.

   set exp = $exposures[$i]
```

```

   if ( -d $exp ) then

      cd $exp
      set exposure_id = $exp

      if ( $verbosity == 1 ) then
         echo 'starting exposure '$exp
      endif

      # --- Make sure we have an empty log file.

      set logfile = $exposure_id'.log'
      if ( -f $logfile ) rm $logfile
      touch $logfile

      # --- If this is not a through run see that some needed files exist:
      #     (In a through run these particular files will not be consulted.)
      #     (This block is here rather than after the "else [no simom.$exp]"
      #      in case a user sets up a simom.$exp file before the first run.)

      if ( $thru == 0 ) then

         # --- Create a sourcecat.dat file if it does not already exist.

         #if ( ! -e sourcecat.dat ) then
         #   if ( ! -e sourcecat.dax ) then
         #      $XMMOMSIMHOME/bin/sourceinit                        >>& $logfile
         #   endif
         #   mv sourcecat.dax sourcecat.dat
         #endif

         # --- Create a tvsourcecat.dat file if it does not already exist.
         #     [Currently disabled.]

         #if ( ! -e tvsourcecat.dat ) then
         #   $XMMOMSIMHOME/bin/tvsourceinit                         >>& $logfile
         #endif

         # --- Create a drift history file if it does not already exist.

         if ( ! -e tdrift.dat ) then
            $XMMOMSIMHOME/bin/mkdrift                               >>& $logfile
         endif

      endif
#

      # --- Execute the sexpose script to run the meat of the simulation.
      #     The sexpose command and arguments are assembled into
      #     an exposure specific simom.$exp script,
      #     which is built if necessary and executed here.

      # --- If there is an exposure-specific simom script, run it.
      #     First edit in the name of the current global_ram file.

      if ( -f simom.$exp ) then

         if ( $verbosity == 1 ) then
            echo 'running existing script: simom'.$exp
         endif

         mv simom.$exp simom.old
         sed -e 's/[0-9]*_ram/'$global_ram_file'/' simom.old > simom.$exp
         source simom.$exp

      # --- If there is not an exposure-specific simom script, build & run it:
      #     Set some good default parameter values & echo them into the script.
      #     Then run the script.  When done, move it to a safe place.

      else

         if ( $verbosity == 1 ) then
            echo 'building a new script: simom'.$exp
         endif
```

```

         set exposure_time = $frames   # number of frames in the exposure
         set frame_time    =      10   # number of seconds per frame.
         set randomseed    =   52731   # seed for random number generator
         set frame_size    =  100000
         set pingpong      =  100000   # ping pong difference, be careful here
         set dump_frame    =   -1000   # don't dump any frame

         if ( -e same_pointing && $i > 1 ) then
            set initframe = 0          # initialize the frame count
         else                          # to  0 if desired (same pointing) or
            set initframe = $iframe    # to -1 if acquisition frame is needed
         endif

         if ($eng_mode >= 0) then
            set sexpose='sexposeng'
         else
            set sexpose='sexpose'
         endif

         echo $XMMOMSIMHOME'/bin/'$sexpose' \'                >  simom.$exp
         echo $working_dir     '        # working_dir \'   >> simom.$exp
         echo $exposure_id     '        # exposure_id \'   >> simom.$exp
         echo $exposure_time   '        # exposure_time \' >> simom.$exp
         echo $frame_time      '        # frame_time \'    >> simom.$exp
         echo $initframe       '        # initial frame \' >> simom.$exp
         echo $randomseed      '        # random seed \'   >> simom.$exp
         echo $dump_frame      '        # dump_frame \'    >> simom.$exp
         echo $thru            '        # thru \'          >> simom.$exp
         echo $global_ram_file '        # global_ram \'    >> simom.$exp
         echo $filter          '        # filter ID \'     >> simom.$exp
         echo $eng_mode        '        # engineering mode'>> simom.$exp

         source simom.$exp
         mv simom.$exp simom.$exp.save

      endif

      # --- Go back to the simulation parent directory.

      cd $parent_dir

   endif

   # --- Increment the exposure counter.

   @ i ++

end
















# --- Clean up

rm /tmp/$global_ram_file
#rm $working_dir/global_ram
#rm $working_dir/local_ram
#rm $working_dir/red_local_ram
rm $working_dir/blue_local_ram
#rm $working_dir/queued_data

exit
```

A.2 sexpose

```
#! /bin/csh
#+ ------------------------------------------------------------------------------
#
# NAME:
#      sexpose
#
# PURPOSE:
#      This script handles all of the tasks required for individual exposures.
#      Tasks that are need once per simulation, regardless of the number of
#      exposures, are handled by the main simom script.
#
# INPUTS:
#      $1 = working directory (location of local_ram files)
#      $2 = full exposure id
#      $3 = number of frames in the exposure
#      $4 = number of seconds per frame.
#      $5 = initial frame (-1 => start with ACQ frame,
#                           0 => start with REF frame)
#      $6 = seed for random number generator
#      $7 = frame dump flag
#      $8 = do a full (0) or thru (1) run
#      $9 = name of global RAM file
#
# CALLS:
#      bluedsp
#      gen_mic1
#      gen_ref_stars
#      icu_command
#      timed_message
#      tm_packet
#      whitedsp
#
# CALLED BY:
#      simom.$exp (which is created and executed by simom itself)
#
# RESTRICTIONS:
#      The environment variable XMMOMSIMHOME must be properly defined.
#
# MODIFICATION HISTORY:
#      04Aug95   ADW   add comment header, update other comments
#      11Jan96   ADW   tidy up CH changes (dump & pingpong mods, some reorg)
#      03May96   ADW   update for changes in idiot.h. etc.
#
# RCS $Id: sexpose,v 1.9 1996/05/10 22:41:31 welty Exp $
#
#- ------------------------------------------------------------------------------

unset noclobber
umask 002
unalias rm
unalias mv
unalias cp

#

# --- Use the RAM files from the working directory (first argument, $1)
#     [Do nothing if the current exposure directory is the working directory.]
#     [Make symbolic links to the *_ram files if this is not the working dir.]

if ( -e global_ram ) then
else
   ln -s /tmp/$9 global_ram
endif

foreach ram (local_ram red_local_ram blue_local_ram)
   if ( -f $ram ) then
   else
      ln -s $1/$ram $ram
   endif
end

# --- Reset the simulation parameters from simom.$exp and set the logfile name.

```

```
set exposure_id   = $2       # specify the full exposure id here.
set exposure_time = $3       # number of frames in the exposure
set frame_time    = $4       # number of seconds per frame.
set iframe        = $5       # initialize the frame count.
set randomseed    = $6       # seed for random number generator
set dump_frame    = $7       # frame dump flag
set thru          = $8       # do a full (0) or thru (1) run
set filter        = $10      # filter ID
set eng_mode      = $11
set pingframe     = 100000
set pongframe     = 14b000
set framesize     = 4b000
set pingpong      = $pingframe  # ping pong difference, be careful here
set logfile = $exposure_id'.log'
# --- Clear the telemetry stream.
if ( -f queued_data ) rm queued_data
if ( $thru == 1 ) set ELECTRONICS1 = $XMMOMSIMHOME/bin/bluedsp
#
# --- The acquisition and reference frames are taken with MIC binning of 2.
set mic_binning = 2
# --- Set the frame start and end times.
set t0 = 0
set t1 = 0
@ t1 = $t0 + $frame_time
# --- If we need to start with an acquisition frame, then:
if ( $iframe == -1 ) then
   echo 'obtaining acquisition frame'
   # --- Set V filter and command the White DSP to run the init_exp swap unit.
   #     NOTE: keep icu_command & countrate consistent!
   $XMMOMSIMHOME/bin/timed_message 'icu_command: set V filter ID'   >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_LOAD_FILT_CONF 6               >>& $logfile
   $XMMOMSIMHOME/bin/timed_message 'icu_command: set exposure ID'   >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_SET_EXP_ID $exposure_id        >>& $logfile
   $XMMOMSIMHOME/bin/timed_message 'icu_command: init exposure'     >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_INIT_EXP $exposure_id          >>& $logfile
   $XMMOMSIMHOME/bin/timed_message 'whitedsp: init exposure'        >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                       >>& $logfile
   # --- Establish the appropriate mic.pin file for the ACQ frame.
   sed 's/Tnext_frame/'$t0'/' mic1.pin.template > delme
   sed 's/Tmic_binning/'$mic_binning'/' delme > mic1.pin
   # --- Generate the absolute pointing (ACQ) frame
   #     and write it into global memory.
   $XMMOMSIMHOME/bin/timed_message 'generate ACQ frame'             >>& $logfile
   if ( $thru == 1 ) then
      # --- If appropriate MIC data already exist in the exposure directory
      #     feed them into the desired simulation electronics.
      #     [What if a data file does not exist? - ADW]
      cat $exposure_id'.mic.'$iframe | $ELECTRONICS1                >>& $logfile
   else
```

```
   set sourcefile = sourcecat_V.dat

   if ( ! -e sourcecat_V.dat ) then
      $XMMOMSIMHOME/bin/timed_message 'run sourceinit'            >>& $logfile
      $XMMOMSIMHOME/bin/sourceinit 6                              >>& $logfile
      $XMMOMSIMHOME/bin/countrate 6                               >>& $logfile
      mv sourcecat.dat sourcecat_V.dat
   endif

   # --- If appropriate MIC data do not already exist, generate them
   #     and feed them into the desired simulation electronics.
   #     Save the data by appending ".-1" to the filename.

   $XMMOMSIMHOME/bin/gen_mic1 \
      $exposure_id $randomseed $t0 $frame_time $sourcefile \
      $eng_mode 6                                                 >>& $logfile
   mv $exposure_id'.mic' $exposure_id'.mic.'$iframe
   mv sourcecat.dot sourcecat_V.dot

   # --- Generate a list of reference stars from the acquisition frame.

   sed 's/Tsourcecat/'$sourcefile'/' \
      gen_ref_stars.pin.template > gen_ref_stars.pin

   $XMMOMSIMHOME/bin/timed_message 'run gen_ref_stars'            >>& $logfile
   $XMMOMSIMHOME/bin/gen_ref_stars                                >>& $logfile

endif

#

if ( $pingpong == $pingframe ) then
  set pingpong = $pongframe
else
  set pingpong = $pingframe
endif

# --- Increment the frame counters and the random seed.

@ iframe ++
@ t0 = $t1
@ t1 = $t0 + $frame_time
@ randomseed += 67

# --- Load the field acquisition parameters into memory.

$XMMOMSIMHOME/bin/timed_message 'icu_command: load ref_stars'     >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_LOAD_REF_STARS ref_stars.cfg    >>& $logfile

# --- Dump out reference star and other relevant info
#     if requested through the -f flag.

if ( ( $dump_frame == $iframe ) || ( $dump_frame >= 1000 ) ) then

   $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM 0 1000              >>& $logfile
   set refdump = $exposure_id'.ref_stars'
   tm_packet < queued_data > $refdump
   rm queued_data

   $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM $pingpong $framesize \
                                                                  >>& $logfile
   set refdump = $exposure_id'_'$iframe
   tm_packet < queued_data > $refdump
   rm queued_data

   if ( $dump_frame == $iframe ) then
     exit
   endif

endif

# --- Command the White DSP to run the field acquisition task.

$XMMOMSIMHOME/bin/timed_message 'icu_command: acquire field'      >>& $logfile
```

```
   $XMMOMSIMHOME/bin/icu_command $IC_ACQUIRE_FLD                        >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: acquire field'           >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                           >>& $logfile

else

   echo 'no acquisition frame requested'

endif

#

# --- Configure the DPU memory for the exposure.

$XMMOMSIMHOME/bin/timed_message 'icu_command: set filter ID'           >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_LOAD_FILT_CONF $filter               >>& $logfile

if ($5 == 0) then    # init stuff needed only if no ACQ frame

   $XMMOMSIMHOME/bin/timed_message 'icu_command: set exposure ID'      >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_SET_EXP_ID $exposure_id           >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'icu_command: init exposure'        >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_INIT_EXP $exposure_id             >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: init exposure'           >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                           >>& $logfile

endif

$XMMOMSIMHOME/bin/timed_message 'icu_command: set exposure time'       >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_SET_EXP_TIME $exposure_time          >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'icu_command: load memory windows'     >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_LOAD_MEM_WDW mmw.cfg                 >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'icu_command: load science windows'    >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_LOAD_SCI_WDW scw.cfg                 >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'icu_command: tracking report on'      >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_REPORT_TRK 1                         >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'icu_command: turn on ref verbose'     >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_ENBL_VERBOSE 0 1                     >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'icu_command: turn on trk verbose'     >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_ENBL_VERBOSE 3 1                     >>& $logfile

echo 'obtaining reference frame'

# --- Update the mic.pin file.

sed 's/Tnext_frame/'$t0'/' mic1.pin.template > delme
sed 's/Tmic_binning/'$mic_binning'/' delme > mic1.pin

# --- Generate the reference frame data if necessary and process them.
#     Note that a REF frame is required for each exposure,
#     even when the telescope pointing has not changed.

$XMMOMSIMHOME/bin/timed_message 'generate REF frame'                   >>& $logfile

if ( $thru == 1 ) then

   cat $exposure_id'.mic.'$iframe | $ELECTRONICS1                       >>& $logfile

else

   set sourcefile  = sourcecat_$filter.dat

   if ( ! -e $sourcefile ) then
      if ( ! -e sourcecat.dax ) then
         $XMMOMSIMHOME/bin/timed_message 'run sourceinit'               >>& $logfile
         $XMMOMSIMHOME/bin/sourceinit $filter                           >>& $logfile
      endif
```

```
      $XMMOMSIMHOME/bin/countrate $filter                           >>& $logfile
      mv sourcecat.dat sourcecat_$filter.dat
   endif

   $XMMOMSIMHOME/bin/gen_mic1 \
      $exposure_id $randomseed $t0 $frame_time $sourcefile \
      $eng_mode $filter                                             >>& $logfile
   mv $exposure_id'.mic' $exposure_id'.mic.'$iframe
   mv sourcecat.dot sourcecat_$filter.dot

endif

#

# --- Command the White DSP to run the choose_guide_stars task.

$XMMOMSIMHOME/bin/timed_message 'icu_command: choose guide stars'   >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_CHOOSE_GS                         >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'whitedsp: choose guide stars'      >>& $logfile
$XMMOMSIMHOME/bin/whitedsp                                          >>& $logfile

if ( ( $dump_frame == $iframe ) || ( $dump_frame >= 1000 ) ) then

      echo $dump_frame $iframe
      $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM 0 8000             >>& $logfile
      set refdump = $exposure_id'_'$iframe
      tm_packet < queued_data > $refdump
      rm queued_data

   if ( $dump_frame == $iframe ) then

      exit

   endif

endif

# --- Increment the frame counters and random seed.

if ( $pingpong == $pingframe ) then
  set pingpong = $pongframe
else
  set pingpong = $pingframe
endif

@ iframe ++
@ t0 = $t1
@ t1 = $t0 + $frame_time
@ randomseed += 67

#

# --- Obtain the frames that constitute the science observation.
#     Do the frame by frame tracking and shift-and-add.
#     We don't want binning for these operations.

set mic_binning = 1

while ( $exposure_time > 0 )

   echo 'frames remaining: '$exposure_time

   # --- Update MIC simulation time in the mic.pin file.

   if ( $iframe == 1 ) then

      sed 's/Tmic_binning/'$mic_binning'/' mic1.pin.template > delme
      sed 's/Tnext_frame/'$t0'/' delme > mic1.pin

   endif

   # --- Generate the tracking frame.

   $XMMOMSIMHOME/bin/timed_message 'generate MIC frame '$iframe     >>& $logfile
```

```

   if ( $thru == 1 ) then

      cat $exposure_id'.mic.'$iframe | $ELECTRONICS1                >>& $logfile

   else

      $XMMOMSIMHOME/bin/gen_mic1 \
         $exposure_id $randomseed $t0 $frame_time $sourcefile \
         $eng_mode $filter                                          >>& $logfile
      mv $exposure_id'.mic' $exposure_id'.mic.'$iframe

   endif

   #

   # --- Dump the frame if desired.

   if ( ( $dump_frame == $iframe ) || ( $dump_frame >= 1000 ) ) then

#       $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM 0 8000         # >>& $logfile
      echo 'dumping' $iframe $pingpong $framesize
      $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM $pingpong $framesize \
                                                                  # >>& $logfile
      set refdump = $exposure_id'_'$iframe
      tm_packet < queued_data > $refdump
      rm queued_data

      if ( $dump_frame == $iframe ) then
#         exit
      endif

   endif

#

   # --- Command the DPU to track guide stars and accumulate the image.

   $XMMOMSIMHOME/bin/timed_message 'icu_command: track guide stars' >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_TRACK_GS 0                     >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: track guide stars'    >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                       >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'reddsp: accumulate image'       >>& $logfile
   $XMMOMSIMHOME/bin/reddsp                                         >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'icu_command: compress & finish' >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_TRACK_GS 1                     >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: compress & finish'    >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                       >>& $logfile

   # --- Decrement the exposure time, i.e., this is a countdown.

   @ exposure_time -= 1

   # --- Increment the frame counters and random number generator seed.

   if ( $pingpong == $pingframe ) then
     set pingpong = $pongframe
   else
     set pingpong = $pingframe
   endif

   @ iframe ++
   @ t0 = $t1
   @ t1 = $t0 + $frame_time
   @ randomseed += 67

end

#

# --- Format the data into valid telemetry packets.
```

```

$XMMOMSIMHOME/bin/tm_packet < queued_data > $exposure_id

$XMMOMSIMHOME/bin/timed_message 'done '$exposure_id                >>& $logfile

# --- Clean up.
#     test -h returns TRUE (0) if the file exists and is a symbolic link.

foreach ram (global_ram local_ram red_local_ram blue_local_ram)
   test -h $ram
   if ( $status == 0 ) then
      rm $ram
   endif
end

rm queued_data *.pin? delme*
```

A.3 sexposeng

```
#! /bin/csh
#+ ---------------------------------------------------------------------------
#
# NAME:
#       sexposeng
#
# PURPOSE:
#
# INPUTS:
#       $1 = working directory (location of local_ram files)
#       $2 = full exposure id
#       $3 = number of frames in the exposure
#       $4 = number of seconds per frame.
#       $5 = initial frame (-1 => start with ACQ frame,
#                            0 => start with REF frame)
#       $6 = seed for random number generator
#       $7 = frame dump flag
#       $8 = do a full (0) or thru (1) run
#       $9 = name of global RAM file
#       $10= filter ID
#
# CALLS:
#       gen_mic1
#       gen_ref_stars
#       icu_command
#       timed_message
#       tm_packet
#       whitedsp
#
# CALLED BY:
#       simom.$exp (which is created and executed by simom itself)
#
# RESTRICTIONS:
#       Make sure that the environment variables XMMOMHOME and XMMOMSIMHOME
#       are properly defined with apporppriate ___.cshrc.
#
# MODIFICATION HISTORY:
#       04Aug95   ADW   add comment header, update other comments
#
# RCS $Id: sexposeng,v 1.2 1996/05/10 22:41:32 welty Exp $
#
#- ---------------------------------------------------------------------------

unset noclobber
umask 002
unalias rm
unalias mv
unalias cp

#

# --- Use the RAM files from the working directory (first argument, $1)
#      [Do nothing if the current exposure directory is the working directory.]
#      [Make symbolic links to the *_ram files if this is not the working dir.]

#if ( -f global_ram ) then
#else
#   ln -s $1/global_ram global_ram
#endif
if ( -e global_ram ) then
else
   ln -s /tmp/$9 global_ram
endif

foreach ram (local_ram red_local_ram blue_local_ram)
if ( -f $ram ) then
else
   ln -s $1/$ram $ram
endif
end

#if ( -f red_local_ram ) then
#else
```

```
#   ln -s $1/red_local_ram red_local_ram
#endif

#if ( -f blue_local_ram ) then
#else
#   ln -s $1/blue_local_ram blue_local_ram
#endif

# --- Reset the simulation parameters from simom.$exp and set the logfile name.

set exposure_id   = $2       # specify the full exposure id here.
set exposure_time = $3       # number of frames in the exposure
set frame_time    = $4       # number of seconds per frame.
set iframe        = -1       # $5       # initialize the frame count.
set randomseed    = $6       # seed for random number generator
set frame_size    = 1000
set pingpong      = 100000   # ping pong difference, be careful here
set dump_frame    = $7       # frame dump flag
set thru          = $8       # do a full (0) or thru (1) run
set filter        = $10
set eng_mode      = $11

set logfile = $exposure_id'.log'

# --- Clear the telemetry stream.

if ( -f queued_data ) rm queued_data

if ( $thru == 1 ) then
   if ($eng_mode == 0 || $eng_mode == 1 || $eng_mode == 2) then
      set ELECTRONICS1 = $XMMOMSIMHOME'/bin/bluedsp_eng'
   else
      set ELECTRONICS1 = $XMMOMSIMHOME'/bin/bluedsp'
   endif
endif

#

# --- Engineering frames use MIC binning of 1 (no binning; for now).

set mic_binning = 1

# --- Set the frame start and end times.

set t0 = 0
set t1 = 0
@ t1 = $t0 + $frame_time

# ---

#if ( $iframe == -1 ) then

   echo 'obtaining frame'

   # --- Command the White DSP

   $XMMOMSIMHOME/bin/timed_message 'icu_command: set V filter'      >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_LOAD_FILT_CONF $filter         >>& $logfile
   $XMMOMSIMHOME/bin/timed_message 'icu_command: exposure ID'       >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_SET_EXP_ID $exposure_id        >>& $logfile
#  $XMMOMSIMHOME/bin/timed_message 'icu_command: load dtw'          >>& $logfile
#  $XMMOMSIMHOME/bin/icu_command $IC_LOAD_MEM_WDW raw.cfg           >>& $logfile
   $XMMOMSIMHOME/bin/timed_message 'icu_command: set eng mode'      >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_ENBL_ENG 1 $eng_mode 1         >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: run eng mode'         >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                       >>& $logfile

   # --- Establish the appropriate mic.pin file for the ACQ frame.

   sed 's/Tnext_frame/'$t0'/' mic1.pin.template > delme
   sed 's/Tmic_binning/'$mic_binning'/' delme > mic1.pin

   # --- Generate the absolute pointing (ACQ) frame
   #     and write it into global memory.
```

```

   $XMMOMSIMHOME/bin/timed_message 'generate HRF frame'           >>& $logfile

   if ( $thru == 1 ) then

      # --- If appropriate MIC data already exist in the exposure directory
      #     feed them into the desired simulation electronics.
      #     [What if a data file does not exist? - ADW]

      cat $exposure_id'.mic.'$iframe | $ELECTRONICS1              >>& $logfile

   else

      set sourcefile = sourcecat_$filter.dat

      $XMMOMSIMHOME/bin/sourceinit $filter                        >>& $logfile
      $XMMOMSIMHOME/bin/countrate $filter                         >>& $logfile
      mv sourcecat.dat sourcecat_$filter.dat

      # --- If appropriate MIC data do not already exist, generate them
      #     and feed them into the desired simulation electronics.
      #     Save the data by appending ".-1" to the filename.

      $XMMOMSIMHOME/bin/gen_mic1 \
         $exposure_id $randomseed $t0 $frame_time $sourcefile \
         $eng_mode $filter                                        >>& $logfile
      mv $exposure_id'.mic' $exposure_id'.mic.'$iframe
      mv sourcecat.dot sourcecat_$filter.dot

   endif

   # --- Dump the frame if desired.

   if ( ( $dump_frame == $iframe ) || ( $dump_frame >= 1000 ) ) then

      #$XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM 0 8000         # >>& $logfile
      echo 'dumping' $iframe $pingpong $frame_size
      $XMMOMSIMHOME/bin/icu_command $IC_DUMP_RAM $pingpong $frame_size \
                                                                # >>& $logfile
      set refdump = $exposure_id'_'$iframe
      tm_packet < queued_data > $refdump
      rm queued_data

      if ( $dump_frame == $iframe ) then
#         exit
      endif

   endif

   # --- Command the White DSP

   if ( -f queued_data ) rm queued_data

   echo 'processing frame'

   $XMMOMSIMHOME/bin/timed_message 'icu_command: set eng mode'      >>& $logfile
   $XMMOMSIMHOME/bin/icu_command $IC_ENBL_ENG 1 $eng_mode 2         >>& $logfile

   $XMMOMSIMHOME/bin/timed_message 'whitedsp: run eng mode'         >>& $logfile
   $XMMOMSIMHOME/bin/whitedsp                                       >>& $logfile

#else

#   echo 'no frame requested - use -a flag'

#endif

echo 'run tm_packet'
$XMMOMSIMHOME/bin/tm_packet < queued_data > $exposure_id

$XMMOMSIMHOME/bin/timed_message 'done '$exposure_id

# --- Clean up.
#     test -h returns TRUE (0) if the file exists and is a symbolic link.

```

```
test -h global_ram

if ( $status == 0 ) then
        rm global_ram
endif

test -h local_ram

if ( $status == 0 ) then
        rm local_ram
endif

test -h red_local_ram

if ( $status == 0 ) then
        rm red_local_ram
endif

test -h blue_local_ram

if ( $status == 0 ) then
        rm blue_local_ram
endif

#rm queued_data *.pin? delme
```

A.4 sim_clean_slate

```
#! /bin/csh
#+ ---------------------------------------------------------------------------
#
# NAME:
#      sim_clean_slate
#
# PURPOSE:
#      This script sets up the DPU simulation memory
#      and initializes the DPU memory area.
#
#      It is equivalent to the following ICU command sequences:
#
#         reset_dsp
#         load_dpuos
#         send_command 0xa42c
#
# INPUTS:
#
# CALLS:
#      runs programs icu_command and whitedsp
#
# CALLED BY:
#      simom
#
# RESTRICTIONS:
#      The UNIX environment variable XMMOMSIMHOME must be defined.
#
# MODIFICATION HISTORY:
#      09Mar95   CH
#      04Aug95   ADW   add comment header, update other comments
#
# RCS $Id: sim_clean_slate,v 1.4 1996/05/10 22:41:33 welty Exp $
#
#- ---------------------------------------------------------------------------

unset noclobber
umask 002
unalias rm
unalias mv
unalias cp

#

# --- Construct a new 20 MB = 5 M words memory area in the current directory.

if ( -f /tmp/$1 ) then
  rm -f /tmp/$1
endif
mkfile 20m /tmp/$1

if ( -e global_ram ) then
  rm -f global_ram
endif
ln -s /tmp/$1 global_ram

# --- If a local_ram file exists, delete it.
#     Absence of local_ram triggers Sun whitedsp to fake loading DPUOS.

if ( -f local_ram ) then
  rm -f local_ram
endif

if ( -f red_local_ram ) then
  rm -f red_local_ram
endif

if ( -f blue_local_ram ) then
  rm -f blue_local_ram
endif

# --- Load DPUOS.

$XMMOMSIMHOME/bin/timed_message 'whitedsp: DPUOS up'             # >>& $logfile
$XMMOMSIMHOME/bin/whitedsp                                        # >>& $logfile
```

```

# --- Initialize:  ICU commands White DSP to initialize.
#                  White commands Red to initialize.

$XMMOMSIMHOME/bin/timed_message 'icu_command: initialize'        # >>& $logfile
$XMMOMSIMHOME/bin/icu_command $IC_INIT_DPU                       # >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'whitedsp: initialize'           # >>& $logfile
$XMMOMSIMHOME/bin/whitedsp                                       # >>& $logfile

$XMMOMSIMHOME/bin/timed_message 'reddsp: initialize'             # >>& $logfile
$XMMOMSIMHOME/bin/reddsp                                         # >>& $logfile
```

A.5 gen_mic1

```
#! /bin/csh
#+ -----------------------------------------------------------------------------
#
# NAME:
#      gen_mic1
#
# PURPOSE:
#      This is a C-shell driver for simulating data production
#      into a photon counter detector.
#
#      The simulation is done in stages:
#         scene:    generate photons from sources in the field of view.
#                   allows moving scene and source.
#         transfer: through whatever medium, air, water, dirt...
#         optics:   telescope, filter, and stuff
#         detector: a photon counting detector, i.e. one that gives x,y and
#                   accurate timing information. Need a lot of specificity.
#         electronics1: this is the simulation of the detector electronics.
#         electronics2: there often are further electronics for data
#                   acquisition and stuff. this simulates that.
#
#      This should serve only as a template. Replace whatever appropriate
#      simulation programs for the specific situation.
#
# INPUTS:
#      $1 = exposere ID string
#      $2 = seed for random number generator
#      $3 = start time
#      $4 = integration time
#      $5 = source catalog file name
#      $6 = engineering mode function
#      $7 = filter ID
#
# CALLS:
#      raygen
#      raytrans
#      platform
#      tracesim
#      mic1
#      bluedsp
#      datapass
#
# CALLED BY:
#      sexpose
#
# RESTRICTIONS:
#      The environment variable XMMOMSIMHOME must be properly defined.
#
# MODIFICATION HISTORY:
#      03Sep93   CH    phocos prototype script
#      07Aug95   ADW   add comment header, update other comments
#
# RCS $Id: gen_mic1,v 1.6 1996/05/10 22:41:28 welty Exp $
#
#- -----------------------------------------------------------------------------

unset noclobber
unalias mv

#

# --- Set the MIC data filename based on the exposure ID.

set exposure = $1
set eng_mode = $6
set filter   = $7
set micdata  = $exposure'.mic'

# --- Define the data chain programs to be used. [datapass is also available]

set XMMOMSIMBIN = $XMMOMSIMHOME/bin

set SCENE        = $XMMOMSIMBIN'/raygen '$filter
```

```
set TRANSFER     = $XMMOMSIMBIN'/raytrans'
set PLATFORM     = $XMMOMSIMBIN'/platform'
set OPTICS       = $XMMOMSIMBIN'/tracesim '$filter
set DETECTOR     = $XMMOMSIMBIN'/mic1'

if ($eng_mode == 0 || $eng_mode == 1 || $eng_mode == 2) then
   set ELECTRONICS1 = $XMMOMSIMBIN'/bluedsp_eng'
else
   set ELECTRONICS1 = $XMMOMSIMBIN'/bluedsp'
endif

# --- Edit inputs into the raygen.pin.template file to make raygen.pin.

sed 's/Trandomseed/'$2'/' raygen.pin.template > raygen.pin
sed 's/Tbegintime/'$3'/' raygen.pin > delme
sed 's/Tintegratetime/'$4'/' delme > delme2
sed 's/Tsourcecat/'$5'/' delme2 > raygen.pin

# --- Build your data chain pipe here.
#     Currently TRANSFER is omitted but the pipe is otherwise complete
#     and output is directed to the MIC data file specified above.

$SCENE    | \
$PLATFORM | \
$OPTICS   | \
$DETECTOR   \
> $micdata

# --- Feed the MIC data into the electronics simulator.

cat $micdata | $ELECTRONICS1

# --- End of gen_mic1
```