A software architecture for realtime data acquisition, instrument control and command processing

Viral V. Tolat
Department of Electrical Engineering
Stanford University
Stanford, Ca. 94305
tolat@nova.stanford.edu

1992 International Telemetering Conference

Abstract

In this paper we describe the flight software for the SETS (Shuttle Electrodynamic Tethered System) experiment. The SETS experiment will fly as part of the TSS-1 (Tethered Satellite System) experiment on STS-46 currently scheduled for July 1992.

The software consists of two major components: the SETSOS (SETS Operating System) and the SETS Application. The SETSOS is a UNIX-like operating system developed especially for realtime data acquisition, instrument control and command processing. The SETSOS, like all operating systems, provides resource management for application programs. It is UNIX-like in that access to resources is provided through a standard set of UNIX system calls. The SETSOS also implements the standard UNIX I/O model and a hierarchical file system. In addition to providing access to physical devices, the SETSOS provides support for two virtual devices: a packet-based data device and a command device. The packet-based data device is used by applications to place data into the telemetry stream. The command device is used to manage commands from the command uplink as well as other sources including other applications and other processors.

The SETS Application is the primary program which runs under the SETSOS to handle data acquisition, instrument control and command processing. It executes as 5 separate processes, each performing a special task. The tasks include housekeeping data acquisition, limit checking, timeline management, and command processing. The processes communicate via shared memory. Time critical processing is coordinated by using signals and interrupts.

In addition to a description of the software, we will discuss the relative merits and tradeoffs of using such a system design for command processing and data acquisition.

1 Introduction

Computer hardware and software for space flight experiments, satellites and other similar systems have traditionally remained as simple as possible in order to maintain a high degree of reliability and fault tolerance. The primary problem with this approach is that much effort goes into the design and development of a single use system. Some tools are developed that can be used in the development of multiple systems, however, for the most part, as each new system is constructed for the special needs of its mission, new effort must be expended and resources used to properly test and verify the system.

With the availability of space qualified microprocessors and other space qualified hardware such as coprocessors comes the ability to use more complex and flexible software that can be easily used for many systems. A key component of this software is the operating system. A flexible and robust operating system can provide a platform for the development of software for a large variety of mission objectives.

Traditional systems have not included what most would call an operating system. Of course there is software to provide resource management, however, this software takes the form of low-level library routines and such. Operating systems have not been used primarily due to their complexity and the overhead introduced by them. In addition, there has been no great need for the functions provided by an operating system. More recently, as the cost of software has out paced the cost of hardware, it is more efficient to use an operating system in order to reduce the recurring cost of development. In addition, the resources and flexibility provided by an operating system can greatly add to the ability to debug and test software and therefore make the software design cycle shorter and more efficient and the final system more reliable.

A key reason to use an operating system is the ability to have a constant platform on which to develop mission software as well as test software and diagnostic software. A key point in deciding on which operating system to use is its ability to operate on multiple platforms and the availability of development tools that can be used with it. The primary reason for this is simply cost. It is much more expensive to do development on flight hardware than on less expensive off-the-shelf systems. This also allows for parallel development effort.

We chose to develop a UNIX-like operating system for three basic reasons. The first is that we like it and in particular, features such as signals, character-based I/O and a hierarchical filesystem. The second reason is compatibility. There are many UNIX systems around that can be used as platforms to test and debug software. The third reason is familiarity. Most of the programmers on the project and most at Stanford know UNIX quite well since it is the primary operating system used on workstations and other research computers on campus. Knowledge of UNIX would alleviate the time and effort to train personnel to code for a new operating system.

Of course, there are some inherent drawbacks to UNIX, primarily its lack of

realtime support, its size, and its availability for an 80186 processor, the target processor for our computer. Given these reasons, we decided to write our own version of UNIX. Our version of UNIX was written, not to be totally compatible with AT&T UNIX or BSD, but to be system call and library call compatible. In addition, our UNIX is small enough to fit in 64k and uses 20k of RAM for data and stack space. No previously written source code was used and was not necessary since only the functionality of UNIX was desired, not its specific implementation.

2 SETSOS

It is accurate to state that the SETSOS is a UNIX-like operating system. From an applications viewpoint, the following aspects of the SETSOS are very similar to that of UNIX:

- I/O model.
- Hierarchical filesystem.
- Signal facilities.
- Process control

From an operating system viewpoint, the functionality provided is implemented in different ways given the constraints under which the the OS was developed. The following constraints are the primary factors which guided the design and implementation of the SETSOS:

1. 64k bytes eprom for the kernel. 64k bytes eprom for mission specific data including application code.
2. Kernel must execute from eprom and require <64k bytes for data.
3. Realtime response to hardware interrupts at the process level.
4. Support for interprocess communication.
5. Support for multiple processors.

The target platform for the SETSOS is an 80186-based computer. Because of the lack of supervisor mode and memory management, many protections customarily provided by OS are not provided. This could be easily remedied in a future version for

a more advanced processor. In addition, due to the lack of suitable secondary storage and adequate hardware support, no virtual memory facilities have been implemented.

As was stated in the introduction, a primary reason for developing a UNIX-like operating system is the desire to carry out flight software development on other hardware platforms and then perform final testing and tuning on the flight hardware. In addition, many people know UNIX and and therefore can develop code without having to learn a new operating system. With this in mind, it was evident that the SETSOS must, from an application programmers viewpoint, behave exactly like UNIX and offer the same functionality. This functionality was achieved by writing the core of the OS with the implementation of a suite of UNIX system calls as an end goal.

In addition to programming functionality, the user interface to the SETSOS is fairly important. A simple shell program similar to the Bourne and C-shells is available for users. The shell provides a command line interface and supports I/O redirection and job control. The shell also supports scripts, however, it does not support many other features common to shells found on most UNIX systems due to a lack of memory.

Given the availability of a large body of information on UNIX, we continue the discussion of the SETSOS by pointing out its differences from UNIX and in particular those features which make it suitable for data acquisition and control in a realtime environment.

2.1 Process scheduling

Process scheduling is first priority-based and then round robin and it is preemptive. In addition, the scheduler is not fair to lower priority processes, i.e., a process with a high priority will starve a process with lower priority. Processes with equal priority get equal access to the CPU. The scheduling interval is 32ms. This is rather short, however, the process switching overhead is small and we have found that this interval works fairly well given our normal process load.

In contrast to our scheduler, most embedded realtime systems use task oriented schedulers that may be preemptive. We use a preemptive scheduler for several reasons but primarily because it is like the UNIX process model. This process model better suits our application architecture than does a task oriented model. In particular, task creation and destruction overhead is far greater than the overhead for context switching and process suspension. Also, the UNIX process model makes it easier to add more processes or tasks to the system and still be assured adequate performance of time critical tasks since additional processes/tasks are automatically relegated to background processing. A UNIX-like process model is essential for a multiuser system and support of application development. And finally, long-lived processes are

convenient mechanism for storing state information during the lifetime of an application.

2.2 Realtime signal delivery

In the SETSOS we differentiate between software signals and hardware signals. Software signals are treated in a manner similar to conventional UNIX, i.e., a signal stack is maintained for a process and when the process is ready to execute, pending signals are serviced. Hardware signals are handled in a more immediate fashion. If a process receives a hardware signal interrupt, the currently running process will be suspended and the signal handler for the process that received the signal will be invoked. Higher priority signals, however, will preempt this process and the appropriate signal handler will be invoked. Upon completion of the interrupt handler(s), the suspended process will be restarted.

This method of dealing with interrupts is similar to the way a conventional embedded system may do so. The overhead incurred to deliver hardware interrupts to a process is relatively small. A small vector table is maintained for each hardware signal. A kernel signal handler first gets the signal and then routes it to upto two processes that have registered handlers for the signal. The overhead incurred is essentially that necessary to perform a context switch. It is up to the programmer to ensure that the signal handler is short in order not to starve out waiting processes. The handler is executed with interrupts enabled and it may be preempted by a higher priority interrupt, however, the handler will not be preempted by the process scheduler.

2.3 Special devices

Two special devices have been created for the SETSOS. The first is used to support command processing, interprocess and interprocessor communication and is called the command device. The second device is used to support telemetry acquisition and processing in a multiprocess environment and is called the burstmode[1] device.

In our system with two cpus, there are three command devices per cpu, one for each cpu (**/dev/cmda, /dev/cmdb**) and one that is for the home cpu (**/dev/cmd**) (see figure 1). For each cpu, there is a single command queue that is accessible via these devices. The cpu specific devices are write-only while the generic cpu device is both readable and writable. The data to be written to the devices is structured so that standard input and standard output cannot be redirected to the command device. The data read and written to the command device consists of 4

[1]This name is historical and is based on the name given to a section of the SETS telemetry page called the burstmode section.

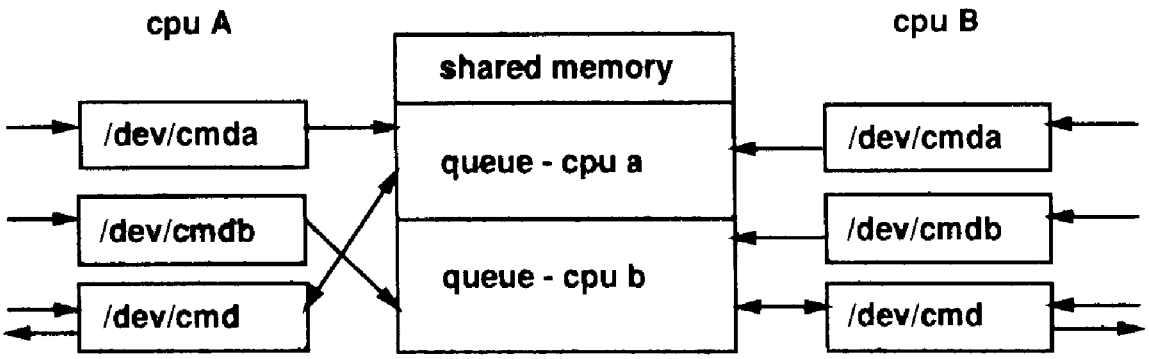

Figure 1: Command device

header bytes and a zero terminated string with a maximum length of 64 bytes. The device queues are stored in shared memory and access is arbitrated via semaphores in the shared memory.

If no process has opened the command device for reading, the kernel checks it once during each pass through the scheduler. If a command is present, the kernel attempts to execute it with the standard input and output set to the kernel's default input and output devices. If a process has opened the command device for reading, it is responsible for processing any commands that are written to the device. As is implied by its name, the command device is used for command processing and in particular it is the mechanism used to direct commands from the ground and from other processes to a master command process.

The burstmode device is accessed as **`/dev/bm`**. The burstmode devices on each cpu are separate and not shared by cpus. The device is only readable by a single process but is writable by many processes. There is only a single burstmode device file, however, there are upto 8 virtual burstmode devices that are writable. When a process opens the device for writing, it is assigned to one of the virtual devices and assigned a default id. The process may then set the id via an ioctl call. The burstmode device accepts a character stream as input therefore allowing the device to be used as standard output for a process.

Reading the burstmode device is rather different. When a process reads from the device, the read returns a structure (packet) of the desired read size. The packet contains the id of the burstmode write, the sequence number and data. Each read from the burstmode device returns a packet from one of the burstmode write devices. Packets are tagged with a sequence number in order to reconstruct a burstmode input stream at the output. The sequence number is added by the device driver. Multiple processes may write to the same burstmode device in which case the data from these processes is interleaved.

Both the command and burstmode device are unique in that they allow multiple writers and a single reader. The devices act as "funnels" for data and commands and we have found them to be extremely useful for our application. These devices are the

primary means of information exchange between processes for the SETS application.

2.4 EPROM/EEPROM/RAM filesystems

The memory that is not used by the kernel for code and data space is structured into a hierarchical filesystem, figure 2. RAM on the cpu board is accessed both through the filesystem **`(/local)`** and directly by processes for code, data and stack space. Half the EPROM, that not used to store the kernel, and the EEPROM bank are structured as

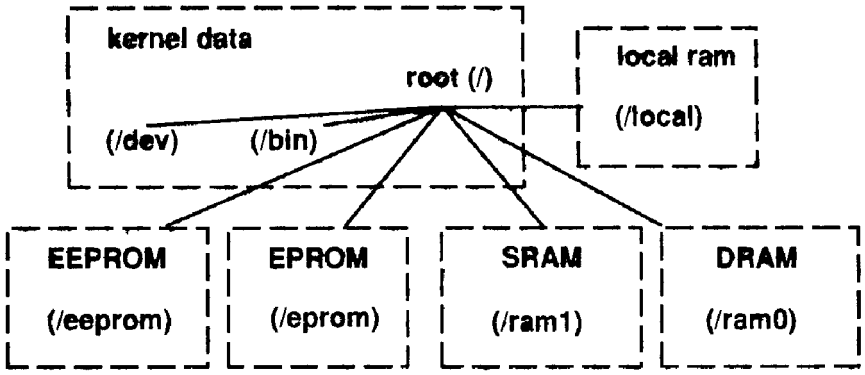

Figure 2: Filesystem structure

a read-only filesystems named **`/sprom`** and **`/eeprom`**. The dynamic and static RAM banks are configured as read-write filesystems, **`/ram0`** and **`/ram1`**, respectively.

The EPROM filesystem and the files in it are created and built on the development system and burned into EPROM along with the kernel. Although the EEPROM is a read-only filesystem, the filesystem can be modified, e.g., files added and removed, with a special application program. The filesystem is read-only in order to prevent erroneous modification of the filesystem and also since a special device driver would be required to write to EEPROM because of its physical characteristics. The local RAM filesystem is created at kernel boot time while the other RAM filesystems are user created and configured with the **`mkfs`** program.

The memory in the system is configured as filesystems in order to make the management of these resources very easy from a users viewpoint. Given the memory management structure of the 80186, much of this memory cannot be conveniently mapped for use as code, data or stack space and therefore using it as a filesystem is the easiest way to manage this resource. Given that writing and reading from solid state memory is quite fast, RAM files can be used to buffer data for a process and accessed with little overhead. In addition, since the data is accessed through the filesystem device drivers, the memory mapping problem is handled by the operating system and not the application. Use of files also allows us to store data and exchange

data conveniently between multiple applications and processes and multiple processors.

2.5 Runtime libraries

In order to reduce application program storage requirements, runtime libraries are used by all programs. The runtime libraries are stored with the kernel in EPROM and are used by the kernel too. The primary difficulty this causes is that whenever the kernel is recompiled and relocated, the address of the procedures in the runtime library change and therefore the programs must be relinked to the libraries and then reloaded into the system.

2.6 Procedural system calls

In contrast to most operating systems, procedure calls are accessed procedurally and not through traps. This is done primarily for speed since a trap has much more additional overhead than a procedure call. Also since the 80186 does not have supervisor mode, it is not necessary to suspend the process in order to change from user mode to supervisor mode before starting the system call. In our kernel, the mode of the system is controlled by a kernel variable which is set before the work in a system call is begun. When this mode variable is set, process switching is disabled so that system calls may run to completion without the process being suspended. Hardware signal interrupts may, however, interrupt a system call. Most parts of a system call are interruptible without a problem and in critical sections interrupts are disabled. Access to slow devices is handled with special process wait states.

2.7 Shared memory support

Shared memory between processes is supported by the availability of a “shared memory” fork system call. This fork is similar to the standard fork except that the process data area is shared by the child and parent process. Since there is no memory management in the 80186 system, processes may also share memory if the address of that memory is known. The shared memory fork, however, will also work in an environment with memory management.

2.8 Multiprocessor support

Support for multiple processors is very basic and is limited to device arbitration and interprocessor communication via the command device. Each cpu is identified by a hostid identifier that is stored in EPROM or may be assigned at runtime. In our system with two cpus, the EPROMS for both cpus are exactly the same except for the hostid.

Command in boot scripts which are used to configure the cpus at powerup may contain an hostid prefix in order to direct that command to one or all cpus. This allows cpus to share boot files and still be be brought up into different configurations.

3 SETS Application

The SETS application is written to make maximum use of the facilities provided by the SETSOS. The SETS application is a single program which creates upto five subprocesses. After creating the subprocesses, the parent process waits for their death or conversely, when it is killed, all the subprocesses die too. The subprocesses perform the following tasks:

1. Command processing and timeline management.

2. Housekeeping data collection.

3. Limit checking.

4. Time management.

5. Data archiving

During nominal operation, the last process, data archiving, is not created since there is no room for data archival during the mission. Instead the process is used as a debugging tool to verify proper functioning of the other processes independent of the communications channel and therefore also to help verify proper operation of the communications channel.

When the SETS application is started, the main process sets its standard output to a burstmode device with an id of 10, therefore all standard output from the process and its children is automatically placed into the telemetry. This output includes all error messages. The limit checking process, after it is created, sets the id of its standard output burstmode device to 11 therefore allowing the ground to easily distinguish out-of-limits fault messages from other messages. Similarly, status information for the limit table, logical table, etc. is output to a burstmode device with different ids therefore making this information easy to extract from the flow of other messages.

In flight configuration, a SETS application is run on each of the two cpus, one as master and one as a slave. See figures 3, 4, and 5 for an illustration of how the different processes interact with each other and the SETSOS.

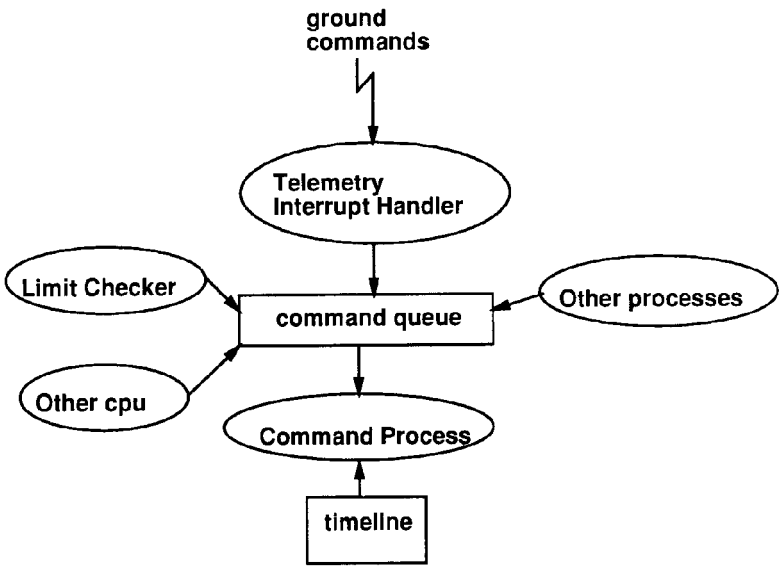

Figure 3: Command processing

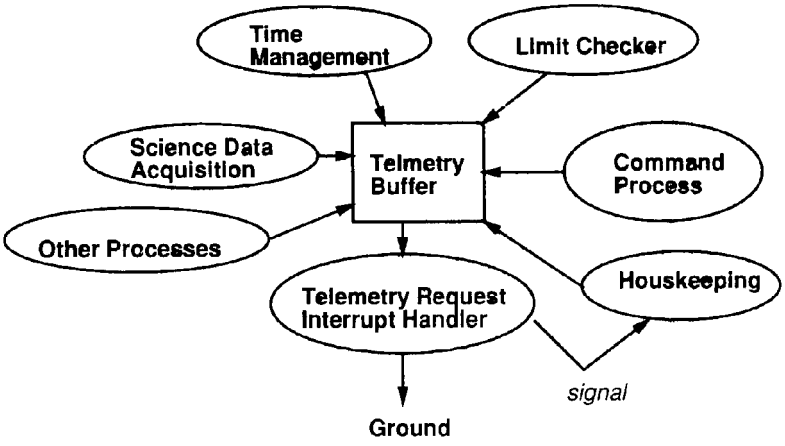

Figure 4: Telemetry Processing

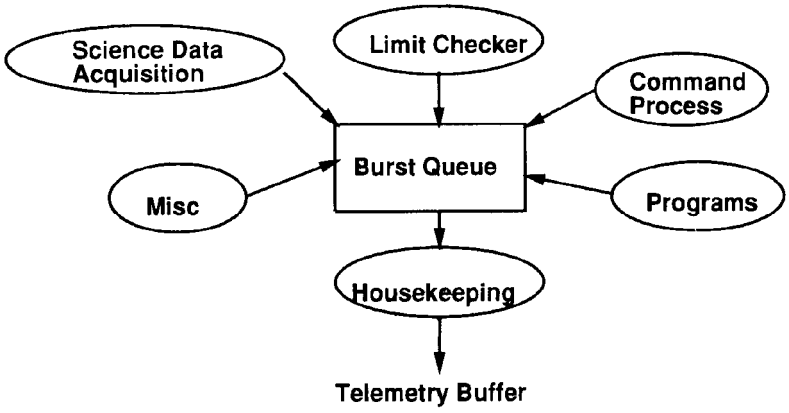

Figure 5: Burstmode Processing

3.1 Command Processing

The first process is responsible for reading commands from the virtual command device and executing them. In addition, it also checks commands in the internal timeline and executes them when they are ready. Commands which may take long time to execute, for example, a command to transmit a data file, are executed by a separate process that is created when such a command is received. This strategy allows the processing of new commands to continue as the other command is processed in the background at a lower priority.

Both the slave and master SETS applications receive all ground commands. A command prefix may be specified which directs the command to a particular cpu or both cpus. By default, commands are only executed by the master application. The cpus are addressed by their host ids.

Sequences of commands may be stored in EPROM, EEPROM or RAM files. Once a sequence is initiated, the command process checks the command device before executing each sequence command so that an external command will be processed immediately. Sequences are processed similar to shell scripts and likewise may be given arguments which can be accessed by commands in the sequence.

An command timeline is implemented to allow the queuing of commands at specific times. The following operations are allowed on the timeline: adding new entries, delete entries, hold entries, hold the timeline, clear the timeline, and restart the timeline. Entries can be added to the timeline by delta time, GMT or MET. The timeline is very useful for implementing command loops. For example, a sequence can be written that places itself in the timeline at its completion to be rerun.

Commands not specific to the SETS application are assumed to be programs and are executed as such. Their standard output is inherited from the command process so that their output is put into the telemetry through the burstmode device. By using the SETSOS shell program, the input and output of an application program may be changed to different file or other device.

3.2 Housekeeping Data

A separate process is used to sample housekeeping data at a rate of once per second. Because the sampling process is synchronized to the transmission of telemetry, the housekeeping process is signaled by the telemetry signal interrupt handler to collect data. After receiving a signal from the telemetry interrupt handler (see figure 4), the housekeeping process samples the housekeeping data and then reads a packet from the burstmode device, if data is available, into the telemetry buffer. The housekeeping process also ensures that if the telemetry acquisition signal is lost, the burstmode device is emptied to prevent the blocking of processes.

3.3 Limit Checking

Limit checking is implemented by a separate process. A limit table contains pointers to telemetry data, low limits, high limits and commands to execute upon an out-of-limits conditions. Commands are available to modify the limits, the action command and which limits should be checked. The limit process wakes up once a second processes the entries in the limit table. On an out-of-limits condition, a the corresponding command is written to the local command device and a fault message is placed in the telemetry through the burstmode device. The limit checker is used as a watchdog for both software and hardware.

3.4 Time management

Management of time related variables is handled by a process that wakes up once a second and modifies time variables in the shared telemetry buffer. The time management process has a low priority so that it does not interfere during periods of heavy commanding. If the GMT signal is lost during the mission, the time management process begins incrementing the GMT based on the clock maintained by the operating system.

3.5 Highrate Science

Highrate science, data sampled at 10Mhz, well above the telemetry downlink bandwidth, must be buffered and then transmitted slowly. This is done by first storing the data in RAM files and then starting a process to write the contents of the file to a burstmode device. This process operates at a low priority therefore not interfering with other operations, but allowing the data to transmit down at a reasonable rate. In operation, the process gets enough cpu time to keep the burstmode buffers full and therefore no telemetry bandwidth is wasted.

3.6 Telemetry processing

Telemetry and other state information is stored in a buffer that is accessible to all processes. Once a second, the telemetry buffer is read into a transmission buffer by the telemetry interrupt handler and then sent to the ground via a highspeed I/O channel. This setup allows all processes to access current telemetry, e.g., the limit and time management processes, and for these processes to easily modify telemetry items as well.

4 Discussion

Once the operating system was debugged and well tested, we had great confidence that any observed errors were due to application software and not a result of system level problems. In addition, hardware problems would be first detected and reported by the operating system.

We found the SETSOS and the capabilities it provided very useful for debugging and verification of the flight software. It was very convenient to be able to quickly write small programs to verify an external interface, for example, or to use extra print statements in the flight code to automatically get diagnostics into the telemetry. During testing of the SETS application, it was simple to make and test modifications since new code could be loaded into an EEPROM file and then executed.

We feel that the use of a preemptive scheduler based on priority is much simpler to use for programmers. In any realtime system, care must be given to the amount of time a process (task) takes and the programmer must be sure that task collisions do not occur. A certain amount of planning must be used to organize the timing of processes and interrupt handlers. In general, hardware interrupts have highest priority and then process priorities are ordered according to function. The programmer must ensure that a high priority process does not take all the cpu time, however, there are cases where it is more important for a process to continue then to be preempted by another process. There are situations where this process model does not allow certain operations to occur at the desired time when collisions arise between high priority processes. Such situations are best avoided, however, the process model performs adequately in such situations and gives cpu time to the higher priority process.

By using the burstmode device for the standard output of all programs, it has been very easy to keep the SETS application small in favor of using separate programs for diagnostics and miscellaneous tasks. Since the overhead to execute a program is relatively small, it takes only a little extra time to use a separate program, however, the memory savings is significant. And more significant is the simplicity of the model. The command device makes is just as simple to have other applications and processes submit commands to the SETS application command process.

It is a challenge in any realtime embedded system to get the timing of processes correct. Hardware interrupts are often used as timing signals when precision and accuracy are required. The ability to have process level hardware interrupts is essential to meeting the timing requirements without writing special operating system level code. Process level interrupt handlers are a very important feature in the SETSOS since slow and nondeterministic responses to external signals are often a criticism for using UNIX systems in realtime applications.

5 Summary

We have presented the design of a flexible realtime embedded system using a UNIX-like operating system. The operating system provides special facilities for operation in a realtime data acquisition and commanding environment. We have used this operating system to implement an application that will control the SETS experiment on STS-46. At the current time, we have gained a year and a half of experience with this system during experiment testing at Stanford University, Utah State University, Marshall Space Flight Center and Kennedy Space Center. As we approach the mission, we are confident that the system will perform correctly and more importantly that we will be able to respond to situations in the mission we cannot as yet anticipate and therefore make the most of the mission.