

4-14-1994

# Optimal kernel development for real-time communications

Monica G. Beltran

*Florida International University*

Follow this and additional works at: <http://digitalcommons.fiu.edu/etd>



Part of the [Electrical and Computer Engineering Commons](#)

---

## Recommended Citation

Beltran, Monica G., "Optimal kernel development for real-time communications" (1994). *FIU Electronic Theses and Dissertations*. Paper 1491.

<http://digitalcommons.fiu.edu/etd/1491>

This work is brought to you for free and open access by the University Graduate School at FIU Digital Commons. It has been accepted for inclusion in FIU Electronic Theses and Dissertations by an authorized administrator of FIU Digital Commons. For more information, please contact [dcc@fiu.edu](mailto:dcc@fiu.edu).

**FLORIDA INTERNATIONAL UNIVERSITY**

**Miami, Florida**

**OPTIMAL KERNEL DEVELOPMENT  
FOR REAL-TIME COMMUNICATIONS**

**A thesis submitted in partial satisfaction of the**

**requirements for the degree of**

**MASTER OF SCIENCE**

**IN**

**ELECTRICAL ENGINEERING**

**by**

**Monica G. Beltran**

**1994**

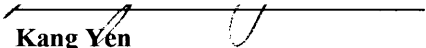
**To: Dr. Gordon Hopkins**

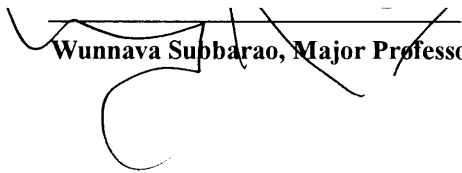
**Dean of the College of Engineering and Design**

**This thesis, written by Monica G. Beltran, and entitled Optimal Kernel Development for Real-Time Communications, having been approved in respect to style and intellectual content, is referred to you for judgment.**

**We have read this thesis and recommend that it be approved.**

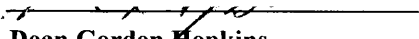
  
**Malcolm Heimer**

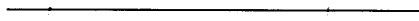
  
**Kang Yen**

  
**Wunnavu Subbarao, Major Professor**

**Date of Defense: April 14, 1994**

**The thesis of Monica G. Beltran is approved.**

  
**Dean Gordon Hopkins**  
**College of Engineering and Design**

  
**Dr. Richard L. Campbell**  
**Dean of Graduate Studies**

**Florida International University, 1994**

I dedicate this thesis to my family. Without their patience, support, love, and understanding, the completion of this thesis would not have been possible.

## **ACKNOWLEDGMENTS**

I wish to thank the members of my committee, Dr. Subbarao, Dr. Heimer, and Dr. Yen, for their helpful comments, patience, and guidance. I would also like to thank Dr. Story and Dr. Roig for their support during my graduate studies. I also want to thank to David Sharp and Lee Hornyak whose knowledge of the communications real-time kernel proved to be invaluable.

A special thanks must go to my major professor, Dr. Subbarao, for his support and constant encouragement.

## **ABSTRACT OF THE THESIS**

### **OPTIMAL KERNEL DEVELOPMENT FOR REAL-TIME COMMUNICATIONS**

**by**

**Monica G. Beltran**

**Florida International University, 1994**

**Miami, Florida**

**Professor Wunnava Subbarao, Major Professor**

The purpose of this research is to develop an optimal kernel which would be used in a real-time engineering and communications system. Since the application is a real-time system, relevant real-time issues are studied in conjunction with kernel related issues. The emphasis of the research is the development of a kernel which would not only adhere to the criteria of a real-time environment, namely determinism and performance, but also provide the flexibility and portability associated with non-real-time environments. The essence of the research is to study how the features found in non-real-time systems could be applied to the real-time system in order to generate an optimal kernel which would provide flexibility and architecture independence while maintaining the performance needed by most of the engineering applications. Traditionally, development of real-time kernels has been done using assembly language. By utilizing the powerful constructs of the C language, a real-time kernel was developed which addressed the goals of flexibility and portability while still meeting the real-time criteria. The implementation of the kernel is carried out using the powerful 68010/20/30/40 microprocessor based systems.

## TABLE OF CONTENTS

CHAPTER	PAGE
<b>I. INTRODUCTION.....</b>	<b>1</b>
Scope of Investigation .....	3
System Overview.....	5
<b>II. SOFTWARE CONSIDERATIONS.....</b>	<b>8</b>
Process Management .....	8
Interprocess Communications .....	14
Memory Management.....	18
Input/Output.....	21
Exception Processing .....	25
<b>III. HARDWARE CONSIDERATIONS .....</b>	<b>28</b>
CISC versus RISC.....	29
Motorola Family of 680x0 Microprocessors.....	34
VMEbus .....	50
<b>IV. ACTUAL MODIFICATIONS .....</b>	<b>57</b>
Initialization .....	58
Scheduling.....	60
System Service Calls .....	63
Memory Management.....	66
I/O Services .....	69
Exceptions .....	74
Time Based Services.....	77
Interprocess Communications.....	78
<b>V. CONCLUSIONS.....</b>	<b>83</b>
Compiler.....	83
Rewriting PCOS.....	86
Problems within the Kernel.....	87
Porting Issues .....	89
<b>VI. RECOMMENDATIONS .....</b>	<b>92</b>
Future Enhancements.....	92
Benchmarking.....	94
POSIX .....	96
C++ .....	100
<b>LIST OF REFERENCES .....</b>	<b>103</b>
<b>APPENDIX .....</b>	<b>107</b>

# CHAPTER 1

## INTRODUCTION

As computer hardware evolved, software was developed to accomplish the task of managing the various computer resources. Software that manages computer resources such as input/output devices, communications lines, memory, and CPU time has generally been termed an operating system.

With the advent of the C programming language, computer programs which would have been traditionally written in assembly language were being developed in C. This migration was a result of C providing both flexibility and portability. The flexibility is projected by C's allowance of bit manipulation and ability to manipulate addressing. Its portability can be seen in that C programs can be compiled on numerous platforms without having to make many modifications if any to the program.

C can be referred to as a system programming language. System programming languages encompass the various languages which allow software engineers to develop system programs. System programs include software such as operating systems, compilers, and interpreters.

The software that has migrated to C has come to include operating systems. Operating systems are responsible for managing and allocating system resources. System resources refer to I/O operations, file and memory management services, and utilities. The software that maintains such hardware devices has generally been written in assembly language, but



with the inception of C, operating systems such as UNIX have been written in C. This allows for the portability of UNIX to many platforms.

Even though operating systems such as UNIX were written in C twenty years ago, designers of real-time operating systems still insist on developing real-time operating systems using assembly languages. A real-time operating system has been defined as a system which responds to external, asynchronous events in a predictable or deterministic amount of time. Real-time operating systems are characterized by the stringent requirements placed on their response time to real-time events. Because of such timing constraints, designers of real-time operating systems believe that only assembly languages will meet the performance levels demanded. These designers feel that high level languages will not meet such high speed requirements. Such reasons may be valid a few years ago, but with the current more optimized compilers and with optimizing coding techniques, real-time operating systems can be written in C.

Currently, real-time operating systems vendors claim to offer real-time operating systems written in C. Upon analysis of their products, one can see that much of their kernels if not all are written in assembly languages while their applications and utilities are written in C.

The kernel of an operating system is referred to as its nucleus. It provides the system administration services such as scheduling and dispatching and services system calls.

If real-time operating systems and especially real-time kernels would be written in C, software designers would be rewarded with software that is modular, portable, and easier to maintain. By having a real-time kernel written in C, there would be additional cost

savings. For instance, as processors keep improving, the transferring of software would be minimal if the kernel is architecture independent.

## **SCOPE OF INVESTIGATION**

This research will reveal that a real-time kernel can be written in C and still satisfy its performance requirements. Such a kernel would be architecture independent.

The real-time kernel that will be analyzed and rewritten in C is PCOS, Process Control Operating System. PCOS is a real-time operating system that was designed in-house to be utilized by NASA for use in its Digital Operational Intercommunications System (OISD). PCOS has used MicroWare's OS-9 as a baseline. In the early 1980's and at the beginning of the OISD project, NASA was investigating various real-time operating systems. It found OS-9 to be the most desirable because of its use of memory modules. At the time, OS-9 had been dedicated to the 6809 microprocessor and had not been ported to the 68000 family. Since NASA had wanted to use the 68000 family of processors, it decided to develop an in-house operating system using OS-9 as a guideline.

NASA has ported PCOS to three platforms: 68010, 68020, and 68030. There are plans to port PCOS to the 68040. Because PCOS has been written in assembly language, a separate version must be maintained for each processor. By maintaining three versions, changes to one version have not always been made to other platforms. For instance, the versions of PCOS for the 68020 and 68030 supported ethernet download one year before

the 68010 version supported ethernet download. Bugs found in the operating systems may be fixed in one version and may not have been fixed in another. By having only one version of the operating system, maintenance will be greatly reduced.

Maintenance will also be simplified because changes to the operating system will be easier to make once PCOS is written in C. All software engineers working on the OISD project are familiar with C, yet less than a third are comfortable with assembly language to make changes or even identify problems within the operating system.

An added benefit of having PCOS be written in C is that PCOS would be more modular and adhere better to the software engineering principle of modularity. Even though PCOS has been developed using assembly language, it is rather modular in design.

C will make PCOS architecture independent. PCOS is currently proprietary meaning that it is married to a specific architecture, namely the Motorola family of 68000 microprocessors. If written in C, PCOS could easily be migrated to other processors such as RISC processors.

The reasons that PCOS developers maintain that PCOS had to be written in assembly language was to maximize speed and minimize size. With good optimizing techniques these reasons are no longer valid.

Even though ADA has been considered a language for real-time applications, it was not chosen for the redesign of PCOS. The applications that have been developed with PCOS and the utilities used in conjunction with PCOS have already be written in C so it seemed logical to redesign the operating system using C. Another reason why ADA was not

considered is that ADA compilers have traditionally lagged C compilers when producing optimizing code. C was also chosen because MicroWare, the company providing NASA with the compiler, does not provide an ADA compiler. The main reason NASA utilizes MicroWare's compilers is that the compilers generate code in a modular format. Another reason why ADA was not chosen is that less than ten percent of the software engineers on the OISD project are familiar with ADA.

## **SYSTEM OVERVIEW**

The PCOS operating system consists of the system level debugger, the kernel, the initialization table, the clock driver, file managers, associated device drivers and device descriptors.

The system level debugger provides three main functions: bootstrap, debug, and BIOS functions. The bootstrap function is called upon a cold start or a system reset. It is responsible for copying vectors to the actual vector table, initializing hardware such as memory management units (MMU) and caches, searching memory for available RAM blocks, and searching for both the kernel module and initialization table. The debug function provides facilities for debugging code such as breakpointing, tracing, and mnemonic disassembly. The BIOS functions provides hardware services to the operating system such as memory management.

The kernel will finish the system initialization using values obtained in the initialization table usually referred to as the init module. System administration and resource management are the responsibilities of the PCOS kernel. The PCOS kernel itself is independent of the hardware since it is mainly a manager. The BIOS functions of the system level debugger and the hardware support services are hardware dependent. The kernel's primary functions include service request processing, module directory maintenance, process scheduling, and exception processing. Service request processing encompasses interprocess communications (IPC) and I/O. The IPC facilities found in PCOS are events, queues, signals, semaphores, and mailboxes. I/O calls are handled by the kernel, but the actual processing is carried out by the file managers and device drivers.

The init module supplies the kernel with configuration information. The parameters that can be found in the init module are the number of process descriptors, path descriptors, queues, semaphores, and mailboxes and offsets to the clock module, queue element table, and table of ports and processes. Table sizes and system device names can also be found in the module.

The clock module provides the software that controls the PCOS system clock hardware. The clock module handles clock interrupts. These interrupts are usually referred to as ticks. With every tick interrupt, the kernel is awakened. During these such interrupts, the kernel determines if time slicing is needed.

The file managers process data streams to and from the device drivers for a similar class of devices. For instance, random-access block structured devices such as disks are supported by the Random Block File Manager (RBF). The Sequential Character File Manager (SCF) handles sequential, character-oriented devices such as terminals and printers. TCP/IP

Ethernet communications are handled by the Ethernet File Manger (ETHER). By grouping similar devices, it is easier to conform to the PCOS standard I/O interface and, thereby, eliminating many unique device operational characteristics.

The device drivers actually perform the basic low-level I/O functions. The device driver is responsible for reading, writing, and initializing the actual device. The device descriptor provides information concerning a specific I/O device such as its logical name, hardware controller addresses, device driver name, and file manager name. Since the file managers and device drivers are concerned with a general class of devices, the device descriptor provides the tailoring of specific I/O functions.

## **CHAPTER 2**

### **SOFTWARE CONSIDERATIONS**

The nucleus of an operating system is the kernel which provides services such as process management, interprocess communications, handling of interrupts, and resource management. The issues related to process management include process scheduling, dispatching, creation, and termination.

### **PROCESS MANAGEMENT**

To understand operating systems, several terms such as process must be identified. A process is the execution of a program or collection of instructions. The term, task, has become interchangeable with the term, process. Although process and task can be considered the same, there is a distinction between multi-tasking and multi-processing. Multi-processing means several processors are executing instructions in parallel, while multi-tasking means several tasks are running concurrently on the same CPU. Multi-tasking and multi-programming can be considered the same.

## Process States

A process can be in any one of the following states:

RUNNING	The process has control of the CPU.
READY	The process is runnable. It is ready to run and is waiting for the CPU to become available.
BLOCKED	The process is suspended waiting for an event to occur.

In PCOS, the blocked state is subdivided into three separate states: waiting, sleeping, and suspended. A process is considered to be waiting if it is blocked until a child process dies. The process is said to be sleeping if it has been blocked for a specific period of time. The suspended state refers to blocking a process which is waiting for an I/O device or for interprocess communications.

A process may switch from one state to another as a result of a service call such as WAIT or SLEEP. Since more than one process can be in the ready, waiting, sleeping, or suspended states, lists are maintained concerning the various states. In PCOS, there is one queue corresponding to each of the ready, waiting, and sleeping states. The suspended state has several queues to support each operation: I/O or interprocess communications.



## **Process Descriptor**

Associated with every process is a process descriptor or process control block. This block of data area provides the following information concerning a process:

process identification number  
parent's process id  
current state  
pointer to local static storage  
pointer to stack area  
process priority

## **Process Creation and Termination**

Because processes are central to the operating system, the system must provide basic functions which handle processes. Such functions are process creation and termination. With the creation of a new process, a new process descriptor, local static storage, and stack area are allocated from a free pool of memory and is assigned. The process which spawns a new process is labeled the parent while the newly spawned process is called the child. In PCOS, a new process is created using the FORK system call. Each new process is assigned two numbers, a process id and an incarnation number. In most operating systems, the process id alone is used for identifying a process. In PCOS, the incarnation number in conjunction with the process id identifies a process because PCOS reuses process id numbers while incarnation numbers are not reused.

Process termination results in deallocating the process descriptor and any other memory used by the process. This memory is returned to the free memory pool. In PCOS, the

termination of a process may be the result of issuing an EXIT system call, receiving a fatal signal, or receiving an exception that has no associated exception handling routine.

## **Scheduling and Dispatching**

The scheduler is responsible for controlling the most valuable resource in a computer system, the processor time. It determines which process is allowed to gain access to the CPU. Once the process is selected to gain access, dispatching is concerned with its activation. Activation mainly refers to moving a process from the ready queue to the run state.

## **Scheduling Algorithms**

Since the scheduler is responsible for the management of the processor time, it aims at providing efficient strategies which consider if a process is I/O bounded or CPU bounded, minimizing response times and overhead and avoiding situations that could lead to starvation and deadlock. Response time includes both the time that a process needs for execution and the time that a process is in the ready queue. Starvation or indefinite postponement means that a process may be ready to run but it does not receive any CPU time. Deadlocks usually develop as a result of multiple processes waiting on dedicated devices. If a process is waiting for an event that never occurs, the result is a deadlock.

Scheduling schemes can be divided into two major groups: non-preemptive and preemptive. A non-preemptive scheduler does not force a process to relinquish control of

the CPU while a preemptive scheduler will take the CPU away from the process. Preemption is crucial to real-time systems because of its demand for fast response times. The cost associated with preemption is overhead.

Originally, schedulers implemented a first come first serve schemes. This scheme is also referred to as the first in first out (FIFO) algorithm. This means that a process would obtain the CPU and would run to completion. Such an algorithm offers a poor response time and could even lead to starvation.

An improvement upon first come first serve technique is the round robin scheme, the preemptive counterpart of the FIFO algorithm. This scheme was designed at improving the response times. With this implementation, each process is given a time slice or quantum. At the end of if each quantum, if a process has not completed its work, it will get interrupted and be placed at the end of the ready queue.

With the above schemes, processes are arranged in the ready queue based upon their arrival and do not have any sense of priority. Priorities provide an indication of criticality of a process. Priorities can be static or dynamic. Static priorities are priorities that do not change while dynamic priorities can change in response to system conditions. Priority scheduling refers to sorting processes in the ready queue according to priority levels.

When certain jobs must be completed within a certain amount of time and priorities are set based upon the deadlines each process must meet, this is referred to as deadline scheduling.

In an effort to further reduce response time, another scheme was analyzed. The shortest job first mechanism grants the CPU to the shortest jobs and allows them to run to completion. This algorithm could very well lead to the indefinite postponement of longer jobs.

The preemptive counterpart of the shortest job first algorithm is the shortest time remaining scheme. The process with the smallest run time to completion is granted the CPU. This scheme has much overhead since it must keep track of the elapsed service record.

To account for the weakness of the shortest job first algorithm, the highest response ratio next strategy was developed. This non-preemptive scheme accounts for service time and the time a job has been waiting for service. With this scheme priorities are issued based upon the ratio of the sum of the waiting time and service time to the service time.

PCOS implements a preemptive scheme in which all processes are ensured processor time. Processes are sorted in the ready queue by age. The age of the process is referred to as the count of how many process switches have occurred since the process' last time slice. When a process enters the ready queue, its age is initialized to the process' priority. This indicates that processes with a higher priority will be placed higher in the ready queue. The process with the highest age is at the head of the queue. Whenever a process is activated, the ages of the other processes in the ready queue will be incremented.

## **Task Switch**

If preemption exists, then enough information concerning the process running on the CPU must be saved such that this process can begin executing as if it had never lost control of the CPU. Since the dispatcher is responsible for the activation of a process that has been granted the CPU, it must save the state of the process using the CPU and then replace it with the state of the new process. The switching of one state to another is referred to as a task switch or context switching. The state of the process may be referred to as its context. Usually, address and data registers, program counters, status flags, stack pointers, and condition codes will provide enough information such that a process could continue executing after an interruption.

## **INTERPROCESS COMMUNICATIONS**

Because real-time applications are often characterized as a set of tasks which are interdependent, the operating system needs to supply services that allow the processes to coordinate and communicate. The blocking of a process until a specific condition is met is referred to as coordination. The transfer of information from one task to another task is considered communication. The services associated with coordination and communications are often called interprocess communications (IPC). Interprocess communications encompass a variety of facilities such as signals, events, semaphores, queues, and mailboxes.

## Signals

Signals provide an asynchronous communications mechanism. A signal is analogous to a software interrupt in that it can cause a process to suspend its execution, execute a specific routine, and then return to the interrupted process. A signal can convey information by means of the signal codes which may have predefined meanings or user defined meanings. The specific routine that services the signal is usually referred to as the signal handler.

The SEND service request is the PCOS system service that is used to send signals from one process or from the operating system to another process. The signal codes that are predefined in PCOS are:

Signal Code	Function
1	Kill
2	Wake-up
3	Keyboard abort
4	Keyboard interrupt
5-65535	User defined

When a signal is sent to a process, the signal code is stored in the process descriptor . The process changes to the active state if it had been in the sleeping or waiting state. Once the process is eligible for execution, the signal is processed. Signals are not queued up. This

means that if a signal is received while a previous signal is being processed, the second signal is ignored.

## **Events**

Events are a synchronous communications mechanism where a user process or the operating system can send a two long words message to any process. The sender identifies the destination process by passing the process id and incarnation number of the target process to the send event service call, `SENDEVNT`. The destination process must explicitly ask for the next event by means of the `GETEVNT` system call. The two long words that are passed to the destination process are referred to as the opcode and address. These values are meaningful only to the destination process but not to the service call. Unlike signals, events are placed in an internal event queue.

The event system is a useful system service where a process must be activated for a variety of reasons. The event queue serves as a funnel through which other system elements notify the process.

## **Semaphores**

This mechanism provides a system service for interlocking between processes. Semaphores are used in the synchronization of concurrent processes which access shared resources such as data or devices. Semaphores do not transmit any information but indicate whether a resource is busy or available. Applications using semaphores for the

coordination of shared resources wait for the resource to become available, mark the resource as busy, utilize the resource, and flag the resource as available.

## **Queues**

The queue system allows processes to exchange data. Such exchange of messages facilitates communication among the processes. A process enqueues a message or collection of elements to a queue using the ENQUEUE system call. A receiving process dequeues a message using the DEQUEUE system service. If a process attempts to enqueue to an already full queue, the process may be placed on a linked list of processes waiting to enqueue to this particular queue. Likewise, if a process tries to dequeue from an empty queue, it may be placed on a linked list of processes waiting to dequeue. The queue mechanism not only provides communication but also coordination between tasks.

## **Mailboxes**

Mailboxes are similar to queues in that they are used for coordination and communication of processes. Mailboxes are essentially queues that hold only one element or mail value. The PCOS system services associated with the mail system are POST and GETMAIL. The POST system call allows a process to place the mail value in a specified mailbox. If the mailbox is full, the process may be suspended on to a linked list of processes waiting to post mail to a mailbox. The GETMAIL system call allows a process to retrieve the mail value from a mailbox. The process may be suspended until there is mail.



## MEMORY MANAGEMENT

The term, memory management encompasses, two functions, memory protection and address translation. The purpose of memory protection is to restrict user programs from accessing memory blocks that have not be specifically allocated for its use. Memory protection reduces the risks of an errant user program affecting the multi-tasking system. The services associated with memory protection are provided by the debugger BIOS services and are not part of the PCOS kernel. The purpose of address translation is to provide greater flexibility of the CPU's addressing modes while ensuring position independence. Address translation allows each user program to believe that it is loaded at address zero. Both of these functions are hardware related because they require the use of a hardware memory management unit (MMU); hence, they are referred to hardware memory management. The hardware also requires the use of the BIOS routines for initialization and support. PCOS requires that all modules be written using only addressing modes that are inherently position independent; thereby, address translation is neither used nor needed.

The term, memory management, can also include the software services which are usually provided by the operating system. These services will include allocating blocks of RAM from a pool of free memory and deallocating the blocks to the pool of free memory when the memory is no longer needed. These services require no hardware support and may be referred to as software memory management.

## **Kernel Memory Management Functions**

During the start-up sequence, PCOS chains all RAM not used for fixed data structures into a free memory linked list which is originated by the debugger boot routines. These fixed data structures include process descriptors, module directory, and queues. Memory can be dynamically allocated from the linked list of free memory in whole number increments of pages. The basic unit of memory allocation is the 256 byte page. By combining contiguous blocks, PCOS will attempt to minimize the number of blocks in the linked list. The automatic allocation of memory occurs with:

- loading of program modules into RAM
- the creation of processes
- processes requesting additional RAM
- I/O system needing buffer space
- system services requiring buffer space

When the memory is no longer needed, it is deallocated and returned to the linked list of free memory. The termination of a process will also lead to the deallocation of its associated data memory.

## **Module Directory**

Any object, such as a program or a data table that is loaded into memory, must utilize the PCOS memory module format. The use of the memory module format allows PCOS to maintain a module directory which contains the name, address, and other related information about each module in memory. A module is added to the module directory

when it is loaded into memory. There is a link count associated with every directory entry. The link count consists of the number of processes using the module. When a process uses the LINK system service to link to a memory module, the link count associated with that particular module is incremented by one. When the process no longer needs the memory module, it unlinks from the module. The associated link count is then decremented by one when the UNLINK system call is used. When the link count reaches zero, its memory is deallocated, and it is removed from the module directory.

### **Memory Module Format**

The memory module format is used in the representation of all code or globally available data under PCOS. Code and variable data must not be mixed within the memory module. These modules do not have to be complete programs; they may contain constants, subroutine packages, or global variables. There are different kinds of memory modules depending on the usage. The requirements imposed on memory modules are that they must be position independent and that they do not modify themselves. Through the use of the module directory, PCOS keeps track of the memory modules.

The following structure pertains to all memory modules: a module header, module body, and check code. The module header provides information which describes the module and its use. Such information includes the size, type, name, attributes, data storage requirements, and execution starting address if it is an executable module. The different types of modules are program, subroutine, data, system, file manager, physical device driver, device descriptor, or user definable. The attributes of the module may include whether the module is executable or re-entrant. The module header is located at the

beginning or lowest address of the module. After the module header, the program or constants section is found. This program or constants section is referred to as the module body. At the end of the module, a three byte Cyclic Redundancy Code (CRC) value is found. Only modules with valid CRCs are listed in the module directory.

## **INPUT/OUTPUT**

PCOS provides a unified I/O system that provides processes with system-wide hardware independent services. The kernel receives all I/O service requests. The kernel does some processing such as allocating data structures for the I/O path. The kernel calls the file managers and device drivers to perform the remainder of the processing. The structural organization of I/O related modules such file managers, device drivers, and device descriptors is hierarchical.

The kernel routes data on the I/O paths from/to processes to/from the appropriate file managers and device drivers. To provide this first level of service for I/O system calls, the kernel maintains two internal data structures: the device table and the path table. When a path is opened, the kernel attempts to link to a memory module having the given or implied device name in the pathlist. This memory module is referred to as the device descriptor which contains the names of the file manager and the device driver associated with the device. By retaining this information, the kernel can route subsequent system calls to these modules.

## **File Manager**

A file manager processes the raw data stream to or from the device drivers for a similar class of devices such that the data will conform to the standard PCOS I/O interface. File managers attempt to remove the unique device operational characteristics. A file manager usually buffers the data stream and issues requests to the kernel for the dynamic allocation of buffer space. It may also monitor or process the data stream such as adding line feed characters after carriage return characters.

PCOS can handle any number of file manager modules. These file managers are re-entrant. One file manager is used for an entire class of devices having similar operational characteristics. PCOS currently support three file managers: the Sequential Character File Manager (SCF), the Random Block File Manager (RBF), and the Ethernet File Manager (ETHER). SCF handles sequential, character-oriented devices such as CRTs and printers. RBF supports random-access devices such as disk drives. ETHER supports TCP/IP Ethernet communications.

## **Device Drivers**

The device driver modules are responsible for performing the basic, low-level I/O transfers to or from a specific type of I/O device. Since these modules are re-entrant, one copy of the module can be used to simultaneously run several different devices which use identical I/O controllers. For instance, the device driver for the MC6850 serial interfaces can communicate to any number of serial terminals that use this chip as an interface.

Each device driver has the following associated routines which are accessed by the corresponding file manager:

- device initialization routine
- read from device
- write to device
- get device status
- set device status
- device termination routine

## **Device Descriptors**

The device descriptor modules are small, non-executable modules which provide information that associates a specific I/O device with its logical name, hardware controller address, device driver name, file manager name, and initialization parameter table. The initialization parameter table consists of values that define the initial state of the operating parameters that are changeable by the PCOS GETSTT and SETSTT system calls. For instance, the initialization parameters associated with a terminal define which control characters are used for backspace or delete.

A device descriptor module must exist for each I/O device in the system. Since both the device drivers and file managers operate on general classes of devices and not specific I/O ports, the device descriptor module tailors the functions of the device drivers and file managers to a specific I/O device. By adding new hardware and creating another device descriptor, additional devices of the same type can be added to the system.

## Device Table

An entry for each active device in the system is contained in the device table. Each entry contains the following information:

- address of the device descriptor
- address of the device driver
- address of the file manager
- number of I/O paths using this device
- starting address of the device driver's static storage
- address of the device

A device is said to be attached if it is represented in the device table. Devices become attached when they are first called upon to perform an I/O function. This corresponds to the first open or create system call executed against a device. The kernel and file managers utilize the device table to provide the necessary path independent information about devices and for device control.

## Path Descriptors

A path descriptor is a data structure that represents every open path. The path descriptor contains information required by the file managers and device drivers to perform I/O functions. Path descriptors are exactly 256 bytes long and are dynamically allocated and deallocated by the kernel as paths are opened and closed.

This structure consists of three sections: a universal section, a file section, and an option

area. The universal section contains parameters such as:

- link to next element in the free list
- mode
- number of open images
- address of the device table entry
- current process ID
- pointer to caller's register stack
- buffer address
- file manager's storage
- path options

The file section is reserved for and defined by each type of file manager for file pointers and permanent variables. The option area is for dynamically alterable operating parameters associated with the file or device. These variables are initialized when the path is opened by copying the initialization table contained in the device descriptor module and can be altered by user programs by means of the GETSTT and SETSTT system calls.

## **EXCEPTION PROCESSING**

PCOS provides services and facilitates the processing of exceptions. An exception is said to alter the normal, sequential flow of program code. Exceptions can be classified as synchronous or asynchronous exceptions.

Synchronous exceptions occur as a result of the execution of an instruction. The occurrence of a synchronous exception is predictable from an examination of the relevant



code, data, and microprocessor state. Synchronous exceptions are often referred to as traps. User traps, which are initiated by the execution of a trap instruction, or illegal instruction traps, which are caused by the processor attempting to execute an invalid instruction opcode, are examples of synchronous exceptions. The SETRAP and TLINK system calls are used to install exception handling routines for user traps. PCOS currently reserves three traps. The three traps are used for system calls processing, use of the Math libraries, and calls to the CIO libraries. The Math libraries include integer and floating point operations, numeric to ASCII conversions, and trigonometric conversions. The CIO libraries feature standard C I/O operations, UNIX system operations, character classification and conversion, and string handling.

Asynchronous exceptions occur as a result of an external event; thereby, they are not predictable from an examination of the system state. Interrupts are often referred to as asynchronous exceptions.

Exceptions may also be classified as fault or non-fault. Fault exceptions generally refer to exceptions which have not installed handlers to deal with the exception, have not initialized interrupting hardware with the appropriate vector, and are a result of an error or illegal operation. The user trap is considered a non-fault exception while the illegal instruction trap is a fault exception. Interrupts are usually considered non-fault exceptions; the spurious interrupt or unacknowledged interrupt is a fault exception. Fault exceptions are broken into two categories: claimed or unclaimed. A claimed fault refers to a fault where a cognizant process takes or is made to take responsibility for the exception. An unclaimed fault is a fault where it is not possible to define a cognizant process to take responsibility for the exception or where the cognizant process has no handler for the occurring fault or is not made responsible for the fault. PCOS allows each process to

install handlers for its own faults with the use of the SETFLT or SETVEC system calls. For a system that must be fault-tolerant, the ability to install a fault handler is of considerable benefit because fault handlers may convert a total failure of the system into a controlled degradation. For instance, if a bus error occurs in a user-installed interrupt handler when attempting to read from a device, the interrupt fault handler gains control. The fault handler is able to determine the error type, access address, and mode for the fault because the handler obtains a copy of the execution stack. The fault handler could set an error flag for the main routine and disable the faulted hardware to prevent future faults. The main routine could even possibly take further action and perform its required task in a different manner.

## **CHAPTER 3**

### **HARDWARE CONSIDERATIONS**

Since real-time systems are so dependent on speed and performance, the hardware used in the implementation of the real-time systems must be carefully considered. The hardware considerations may include analyzing what type of processor is best suited for real-time systems or evaluating a bus that is well suited for real-time applications. Currently, there are many microprocessors to choose from, but they generally fall under two major categories: Reduced Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC). The hardware analysis will evaluate the advantages and disadvantages associated with these major categories: RISC and CISC. In its real-time implementation of the communications system, NASA opted for CISC and more specifically, the Motorola family of 680x0 processors. At the time of its decision the only viable option was CISC because RISC technology was not even available for consideration at the time. Since NASA chose the Motorola family of 680x0 processors, the bus that was best suited for real-time and interfaced quite well with the 680x0 processors was the VMEbus. An overview of the 680x0 family of processors and the VMEbus will be also provided in this hardware analysis.

## CISC VERSUS RISC

Microprocessor technology is divided into two classes: RISC and CISC. RISC is based on the philosophy that overall performance can be maximized if the instruction set is reduced to the simplest instruction which can be executed one or more per clock cycle. The RISC processors rely on large register sets or register windowed architectures and pipelined or super-scalar implementations in order to achieve processing speed. The RISC chips usually have a lower transistor count than CISC chips, but the transistor count increases if I/O features or caches are incorporated. RISC is dependent on the software for complex computations; this means that optimizing compilers are needed. RISC is the newer technology while CISC is the more established technology. RISC was developed in the late 1980's while CISC has been around since the development of the microprocessor. CISC processors are considered general purpose processors while RISC were originally targeted for a workstation environment. Since CISC is characterized by complex instruction sets, it requires more execution time. CISC also tends to require more chip area. CISC processors are becoming more RISC-like through pipeline implementations and faster execution times. Since CISC offers more complex instruction sets, each CISC instruction accomplishes more than a RISC instruction and uses more sophisticated addressing modes. With CISC technology, register sets are kept small and parameters are passed on the stack.

## Hardware Issues

Since real-time systems require determinism and low interrupt latency, developers must determine how hardware features such as interrupts, register sets, caches, MMUs, pipelines, and floating point units can adversely affect the requirements for determinism and latency.

Real-time systems require efficient interrupt support. RISC processors generally handle interrupts in software. These processors usually only have a single interrupt vector with no hardware support for multiple interrupts. CISC processors do offer hardware support for multiple interrupts.

Context switching times and interrupt handling times are affected by the amount of registers present in the system. Since RISC processors have large register sets, saving all the registers during a context switch or in response to an interrupt can greatly increase context switching or interrupt handling times. By taking advantage of compiler optimizations or register windowing, not all registers must be saved. If register windowing results in a window overflow, this could be very costly in terms of time. Since CISC processors have small register sets, the saving of registers is not as time-consuming as with the RISC technology.

Both RISC and CISC technologies are using caching. Caches generally reduce the average execution time, but with real-time systems, the concern is with the worst case time and not the average time. The worst case refers to having cache misses with every instruction and data fetch. Caching can have adverse effects on determinism due to cache misses, refilling, cache thrashing, cache flushing and invalidation. To overcome such

effects a cache, that can lock in code and data, should be used. Another effect associated with flushing or invalidation is an increase in interrupt latency time because interrupts need to be locked during the flushing or invalidating. Although both RISC and CISC use caches, RISC tends to have more misses as a result of the increased instruction rate. For systems where multiple CPUs or DMAs exist, cache coherency becomes an issue. Bus snooping hardware is considered the best solution to the cache coherency issue.

Memory Management Units (MMU) are also used in both technologies. MMUs can be used for translating virtual addresses into physical addressing, memory protection, tracking the memory pages, and enabling and disabling the cache for different parts of memory. For real-time systems, many of these features are not needed. Since real-time systems avoid page swapping, there is no need for address translations or tracking of pages. Also memory protection is generally not needed in real-time systems with the exception of fault-tolerant applications. Table walks and page faults also affect determinism and throughput. In order to overcome these effects, translation look-aside buffers that are lockable should be used. The table look-aside buffers are responsible for caching address translations and is the major culprit in affecting determinism. When analyzing memory systems, trade-offs are often made between performance, determinism, and latency.

The use of floating point units (FPU) may increase context switching times if FPU registers are also saved. Optimization can be used in order to save the FPU registers in cases where both the interrupted task and the interrupting task utilize the FPU. Trade-offs may be made regarding the use of an FPU versus emulation libraries. FPU are able to perform complex functions in hardware, yet they increase context switch time, space, and power consumption.

## Software Issues

When selecting a type of processor not only are the hardware issues analyzed, but the analysis must include what software issues are affected by the processor selection. These issues include: the requirements imposed on the operating system, compilation, the availability of tools, compatibility, and the size of the code.

Real-time applications rely heavily on the operating system regardless of the CPU that is selected, but the RISC processors tend to need even more support from the operating system. RISC chips need an operating system which can handle interrupts in software.

Although optimizing compilers are always important, they become even more critical with RISC architectures. RISC processors need compilers which offer compiler register allocation and an instruction rescheduler. Compiler register allocation refers to not having to save compiler-saved registers upon receiving or restoring interrupts due to the compilation environment guaranteeing that these registers are saved across procedure calls. The instruction rescheduler reorganizes the code to make more use of the pipeline or multiple instruction units.

Since CISC is the more established technology, more development tools are available for the CISC processors than for the RISC chips. Development tools include computer aided software engineering (CASE) tools, simulators, language support, shells, debugging tools, and analysis tools.

Compatibility becomes an issue if there is a need to run existing binaries without recompilation when upgrading. With CISC technology, upgrading within the same family

does not require recompilation because of object code compatibility. Changing from one family to another would require recompilation regardless of the technology.

Code size may be relevant in the decision analysis. Since RISC offers a simpler instruction set, it requires more instructions to do comparable work. RISC programs tend to be 25-100% larger than equivalent CISC programs.

The selection of the processor technology essentially depends on the application. The two technologies should be analyzed in terms of how well each meets the needs associated with a particular application. Trade-offs will be made as a result. The most attractive feature of the RISC processors is its performance while its handling of frequent context switching needs attention. Real-time applications such as medical instrumentation, radar and satellite require performance, yet these applications do not have a need for frequent context switching, making them candidates for RISC technology. In applications where frequent context switches are needed and considered more important than just performance, CISC technology proves to be the better choice. CISC offers the ability to handle frequent context switching and interrupt handling better than RISC technology. Applications where CISC tends to be used include industrial control applications and communications.



## MOTOROLA FAMILY OF 680X0 MICROPROCESSORS

The 680x0 family of microprocessor is extensively used not only in real-time applications such as telecommunications switching and voice/data transmission, but in CAD/CAM systems and artificial intelligence. The 680x0 family provides upward compatibility, a powerful instruction set, and flexible addressing modes. Features such as these have contributed to the popularity of the Motorola family of 680x0 processors. Another benefit of using the Motorola family is that there is a large selection of peripherals that complements the microprocessors. Peripherals which provide memory management, data communications, Direct Memory Access (DMA) control, network control, general I/O and graphics can be easily integrated with the 680x0 family.

The Motorola family of 680x0 microprocessors was chosen by NASA as the platform for its real-time application. The real-time communications system uses the 68010, 68020, and 68030 processors, but there is a strong possibility of migrating to the 68040 and even the 68060.

An overview of the various members of the 680x0 family of processors will be provided. Motorola started with the 68000 and progressed to the 68010 through 68060 as features were implemented.

## 68000

The MC68000 provides 8 32-bit address registers (A0-A7), and 8 32-bit data registers (D0-D7), a 32-bit user stack pointer (USP), a 32-bit supervisor stack pointer (SSP), a 32-bit program counter (PC), and a 16-bit status and control register (SR). Even though the address registers and data registers are 32-bit wide, the internal data bus on the MC68000 is 16-bit, thereby, making it a 16-bit microprocessor. The internal address bus is 24-bit and has 16 Megabyte direct addressing range.

Although the MC68000 provides all these registers, there are some restrictions. For instance, the user stack pointer is really A7. Another restriction is that the supervisor stack pointer and the upper byte of the status register are only available if the microprocessor is configured in supervisor mode. The upper byte of the status register contains the interrupt mask, trace mode (T), and supervisor state bit (S). The lower byte of the status register contains condition codes such as extend (X), negative (N), zero (Z), overflow (V), and carry (C). As seen, the microprocessor is capable of running in two different modes, user and supervisor, depending on the supervisor state bit. The user mode applies to all application software while the supervisor mode can be used for operating system software.

The MC68000 provides 56 powerful instruction types that include math, branching, addressing, and logical functions. The MC68000 can support five basic data types and 14 addressing modes. The five data types are bits, BCD digits (4-bits), bytes (8-bits), words (16-bits), and long words (32-bits). The 14 addressing modes supported by the MC68000 are based on the following six basic types: register direct, register indirect, absolute, program counter relative, immediate, and implied.

Vectored interrupts and seven priority levels with level seven being the highest priority are features that make the MC68000 suitable for real-time applications. The processing state associated with the use of these interrupts is the exception processing. The MC68000 can be in any of the three processing states: normal processing, exception processing, and halted processing. The normal processing state refers to the execution of instructions where memory references are to fetch instructions and operands and to store the results of the operations. The exception processing deals with interrupts, trap instructions, tracing, and any other exception conditions. Exceptions can be internally or externally generated. For instance, an unusual condition such as an illegal instruction may arise in the execution of instructions resulting in an internally generated exception. An example of an externally generated exception may be a bus error or an interrupt. The purpose of exception processing is to provide an efficient context switch so that the processor is capable of handling unusual conditions. The halted processing state indicates catastrophic hardware failures. If a bus error occurs while processing a previous bus error, the result is a halted processor.

There are a variety of input and output signals associated with the MC68000 processor. The input and output signals consist of the address bus, the data bus, asynchronous bus control, bus arbitration control, interrupt control, system control, M6800 peripheral control, processor status, and clock. The address bus is a unidirectional three-state bus which addresses 16 Megabytes of data. The data bus is a bi-directional three-state bus used in the transfer of data. The lower 8-bits of the data bus, D0-D7, are also used by the external devices during an interrupt acknowledge cycle as a depository for the vector number. The asynchronous bus control consists of an address strobe, read/write, upper and lower data strobes, and a data transfer acknowledge. The asynchronous bus control

uses its control signals to handle asynchronous data transfers. The bus arbitration control provides a means of determining which device is to be bus master. The control signals that provide the bus arbitration are bus request, bus grant, and bus grant acknowledge. The interrupt control consist of three input pins which indicate the encoded priority level of the interrupting device. The system control signals are used for resetting, halting the processor or indicating that a bus error has occurred.

The M6800 peripheral control signals are utilized when interfacing with synchronous M6800 peripheral devices. These peripheral signals include an enable, valid peripheral address, and valid memory address. Examples of the M6800 peripheral devices include MC6821 Peripheral Interface Adapter, MC6840 Programmable Timer Module, and MC6843 Floppy Disk Controller.

The processor status consists of three function codes output lines, FC0, FC1, and FC2, which reflect the state of the processor and the cycle type currently executed. The function code output are user data, user program, supervisor data, supervisor program, and interrupt acknowledge.

The clock signal is a TTL-compatible input which is internally buffered and used for the development of internal clock signals.

The MC68000 utilizes memory mapped I/O. Memory mapped I/O refers to when a given memory address is used as an input or output port address. This means that the existing address, data, and control buses can handle I/O transactions as if they were ordinary memory accesses. Thereby, memory mapped I/O does not require any overhead such as special I/O instructions. The penalties associated with memory mapped I/O are that the

I/O transfer must be of the same width as a normal memory access and some address space must be dedicated to I/O space.

There are three types of cycles that describe the data transfers: the read cycle, the write cycle, and the read-modify-write cycle. The read cycle consists of the processors receiving data from the memory or a peripheral device. The write cycle sends data to either memory or a peripheral device. The read-modify-write cycle consists of reading the data, modifying the data in the arithmetic logic unit, and writing the data back to the same address. This cycle is indivisible, meaning that the address strobe is asserted throughout the entire cycle. The test and set (TAS) instruction uses such a cycle for the interlocked multiprocessor communications.

## **68010**

The MC68010 is fully user object code compatible with the MC68000 and pin for pin compatible with the MC68000. What the MC68010 offers, that the MC68000 does not, is the virtual memory support and an enhancement in the timing of instruction execution.

The MC68010 still maintains the same register set as the MC68000 used in the user programming model. The MC68010 still offers eight 32-bit data registers, D0-D7, and eight address registers, A0-A7, where A7 is also used as a stack pointer. A 32-bit program counter and a 16-bit status register where the upper eight bits can only be accessed in the supervisor state are also part of the MC68010. The MC68010 unlike its predecessor offers a 32-bit vector base register, VBR, and two 3-bit function code registers, SFC and DFC, to supplement the supervisor programming model. The VBR is

used in the calculation of the actual address of the exception vector. When an exception occurs, the exception vector from the appropriate location in the vector table is obtained and is summed with the contents of the VBR yielding the actual address of the exception vector. The VBR allows for the support of multiple exception vector tables. The alternate function code registers are SFC, source function code, and DFC, destination function code. The alternate function code registers are used only in systems with memory management units that distinguish between user address space and supervisor address space. If the source operand is in memory then the SFC is used in determining the address space. The DFC is used in determining the address space if the destination operand is in memory.

The MC68010 also maintains a 16-bit data bus and a 24-bit address bus capable of addressing 16 Megabytes of memory. With the MC68010, 16 Megabytes of physical memory need not be present for the user program to access it. With virtual memory techniques, it will appear to the user program as if the 16 Megabytes of memory are present when in fact only a small amount of physical memory may be present. A user program may be developed to use a large amount of memory while really only using a small amount of physical memory. Virtual memory systems also allow user programs to access devices that are not physically present. Proper software emulation will make the physical system appear to the user program as a complete computer system, thereby, giving full access to all emulated resources. The MC68010 supports virtual machine techniques through its instruction continuation mechanism. Accessing memory that is not present would result in a bus or address error. If an address or bus error occurs, the processor will push its internal state unto the supervisor stack. The appropriate exception handler is erased. Upon completion, the MC68010 will reload its internal state and resume execution.

The MC68000 offers 56 instruction types while the MC68010 provides 57 instruction types in the user environment. The new instruction offered in the user mode is to correct a flaw that exists in the MC68000 where a user program can access the status word using the MOVE SRE,<ea> instruction. The new instruction, MOVE CCR,<ea>, allows the user program to access the condition code register byte. The new instruction type offered in the supervisor mode is the instruction, Return and Deallocate, RTD. The RTD instruction will do the following: pull the long word at the top of the stack, place this long word in the program counter, and increment the stack pointer by a designated number of bytes.

Although the MC68010 has essentially the same instruction set as the MC68000, the MC68010 can execute some instructions faster than the MC68000. This is due to improvements in the internal microprogramming of the MC68010. Improvements in the performance of the MC68010 are also a result of its loop mode operation. The MC68010 has a two word, tightly coupled instruction pre-fetch mechanism. Because of this when the loop mode is entered, instructions are executed only once when the loop is first executed. In the loop, only data accesses are performed. The loop mode is entered when the instruction is a loopable instruction, a DBcc instruction, or a branch displacement to the loopable instruction.

Since the MC68010 is pin for pin compatible with the MC68000, the signal inputs and outputs are essentially the same. Like the MC68000, the MC68010 has the following signals: an address bus, data bus, asynchronous bus control, bus arbitration control, interrupt control, system control, M6800 peripheral control, processor status and a clock. The slight difference is associated with the usage of the processor status and the interrupt

control. In the MC68000, the processor status function code outputs, FC0, FC1, and FC2, when asserted high, were used as an interrupt acknowledge. With the MC68010, the interrupt control relies on the function code outputs and A16-A19 being asserted. The function code outputs when asserted high refer to the CPU space cycle in the MC68010. The CPU space cycle is an extension to the cycles that were already used in the MC68000.

The MC68010 now has four data transfer cycles as opposed to the three cycles which define data transfers in the MC68000. The read cycle, write cycle, and read-modify-write cycle in the MC68010 behave exactly as that of the MC68000. The CPU space cycle refers to either reading a peripheral device vector number or a breakpoint instruction. A cycle where a vector number is read is referred to as an interrupt acknowledge cycle. The breakpoint read cycle is executed in response to a breakpoint instruction.

## **68020**

The MC68020 is the evolution of the Motorola family of 68000 processors into a 32-bit microprocessor. The MC68020 provides 32-bit address and data buses while still maintaining object code compatibility with its predecessors. The features of the MC68020 include new addressing modes for the support of high level languages, new data types, fast on-chip instruction cache, coprocessor interface, and 4 Gigabytes of direct addressing.

The MC68020 can be subdivided into two main sections: the bus controller and the micromachine. The bus controller schedules the bus cycles for operand or instruction accesses. The bus controller contains the address pads, data pads, and instruction cache. The micromachine contains the execution unit, nanorom and microrom storage, an



instruction decoder, an instruction pipe, and associated control sections. An address section, an operand address section, and data section constitute the execution unit. The nanorom and microrom storage provide the microcode control. Instruction decoding and sequencing of information is provided by the Programmable Logic Arrays (PLA). The secondary decoding of instructions and the generation of the actual control signals is provided by the instruction pipe and associated control sections.

Like its predecessors, the MC68020 offers 8 32-bit data registers, 8 32-bit address registers, a 32-bit program counter, and a 16-bit status register. The MC68020 also provides the VBR and alternate function code registers, SFC and DFC, that were introduced with the MC68010. In addition to these registers, the MC68020 offers 2 32-bit supervisor stack pointers and 2 32-bit cache handling registers. The supervisor stack pointers are the interrupt stack pointer (ISP) and the master stack pointer (MSP). In order to increase the efficiency of a multitasking operating system, the MC68020 separates the supervisor stack space associated with the user tasks from the stack space pertaining to interrupt related tasks by maintaining two stack pointers. The ISP maintains the stack area for interrupt associated activities while the MSP handles the tasks related to the supervisor stack area. Such an arrangement allows for the separation of supervisor task related activities from asynchronous supervisor I/O related activities. The cache handling registers are the cache control register (CACR) and the cache address register (CAAR). The CACR provides the control and status accesses to the instruction cache. The CAAR contains the addresses of the cache control functions which require an address. Another difference associated with the register set of the MC68020 is the addition of two control bits in the status register. These two bits are an additional tracing bit and a Master/Interrupt state bit. All members of the 68000 family offer instruction

tracing, but the MC68020 also offers the capability to trace only the change of flow instructions such as branches or jumps.

The MC68020 has an increased instruction set to maximize use of the 32-bit bus and to support high level language structures and operating systems. Enhancements to the instruction set include upgrades to bit field operations, PACK and UNPACK operations for BCD conversions, bound checking at both upper and lower limits, additions to the system trap facilities, compare and swap instructions, module support with the call module (CALLM) and return from module (RTM) instructions.

The MC68020 offers 18 addressing modes which include the following 9 basic types: register direct, register indirect, register indirect with index, memory indirect, program counter indirect with displacement, program counter indirect with index, program counter memory indirect, absolute, and immediate. The register indirect addressing modes support capabilities that are useful for handling advanced data structures. These facilities include post-increment, pre-decrement, offset, and indexing. The program counter relative mode also offers the indexing and offset capabilities which are used to support position independent software. In addition to these addressing modes, the MC68020 offers data operand sizing and scaling.

The MC68020 supports 7 basic data type. These data types are bits, bit fields, BCD digits, byte integers (8 bits), word integers (16 bits), long word integers (32 bits), and quad word integers (64 bits).

The coprocessor interface extends the instruction set to include new data types and operations which are implemented by the MC68881, floating point coprocessor. This

interface, thereby, provides full support for the sequential non-concurrent instruction execution model. This means that the executing of instructions in the coprocessor does not occur simultaneously with the execution of instructions in the main processor unless it can be properly accommodated. For instance, the coprocessor will allow the MC68020 to proceed executing an instruction while the MC68881 continues a floating point instruction up to the point where the MC68020 sends another request to the MC68881. The coprocessor is operated through the coprocessor interface registers which are accessed through normal asynchronous bus cycles. Because of this, predecessors of the MC68020 can utilize a coprocessor. The advantage of utilizing the MC68020 is that a communications protocol is offered in hardware and microcode. The MC68020 also handles all operations automatically, thereby, allowing the programmer to use the instructions and data types provided by the coprocessor as extensions. The predecessors of the MC68020 require software emulation to handle the coprocessor and do not provide any hardware support.

In order to reduce the external bus activity and to increase the effective CPU throughput, the MC68020 provides an on-chip 256 byte instruction cache. By increasing the effective CPU throughput, the MC68020 compensates for large memory sizes or cheaper slower memory which may have increased average access times. An important benefit of the cache is that it allows instruction stream fetches and operand accesses to proceed in parallel.

Like its predecessor, the MC68020 maintains the following signals: data bus, address bus, asynchronous bus control, interrupt bus control, function codes, bus arbitration control, system control, and clock. The data and address buses are increased to 32-bit each. The MC68020 also provides the following signals: cache control and transfer size. The

transfer size indicates the number of bytes that remain to be transferred for the current bus cycle.

## **68030**

The MC68030 is the second generation 32-bit microprocessor. The MC68030 is based on the MC68020 but with more enhancements. The enhancements include a versatile bus controller, an additional 256 byte data cache, on-chip memory management unit (MMU), and pipeline architecture. The bus controller supports two-clock cycle bus accesses and one-clock cycle burst access which will maximize performance with paged mode, nibble mode, static column DRAM technology. The MC68030 offers both a 256-byte on-chip instruction cache and a 256-byte data cache which further boost performance. These two caches can be accessed simultaneously. The on-chip memory management unit provides zero translation time to any bus cycle and reduces the minimum physical bus cycle to two clocks. The pipeline architecture allows for multiple instructions to be executed concurrently and for accesses from the internal caches to occur in parallel.

The MC68030 maintains the rich instruction set and addressing modes of the MC68020. The new instructions introduced by the MC68030 are associated with communicating with the paged memory management unit. The seven basic data types and the eighteen addressing modes of the MC68020 are also supported.

The MC68030 maintains the same user programming model such as the 8 address registers, 8 data registers, a user stack pointer, the program counter, and the condition code register. The MC68030 provides extensions to the supervisor programming model

of the MC68020 as a result of having the memory management unit. The new registers are: a 64-bit CPU root pointer register (CRP), a 64-bit supervisor root pointer register (SRP), a 32-bit translation control register (TC), a 32-bit transparent translation register 0 (TT0), a 32-bit transparent translation register 1 (TT1), and a 16-bit MMU status register (MMUSR). The CRP contains a descriptor for the first pointer to be used in the translation table search for page descriptors associated with the current program. The SRP is used as a pointer to the translation tables for all supervisor accesses if the Supervisor Root Pointer Enable (SRE) bit is set in the TC. If the SRE is not set, the CRP is used for both supervisor and user translations. The TC is used in the configuration of the table look-up mechanism, page size, and initial shifts of logical addresses. The TC also contains the enable bit which enables the MMU. The transparent translation registers are used to define two transparent windows for transferring large blocks of data with untranslated addresses. The MMUSR provides status information related to a specific address translation. The status information may be useful in locating causes of MMU faults.

The MC68030 provides the same signal groups as its predecessor, MC68020, but it also provides emulation support.

## **68040**

The MC68040 expands features provided by the MC68030. The MC68040 also makes further improvements in performance. The MC68040 provides two independent memory management units. One MMU is for instruction accesses, and the other one is for data stream accesses. Like the MC68030, the MC68040 also offers two independent caches,

an instruction cache and a data cache, but the size of the caches has increased to 4 K each. The MC68020 and the MC68030 both offered a coprocessor interface, but the MC68040 provides an on-chip floating point unit. Performance enhancements also have resulted from a high degree of parallelism. The MC68040 utilizes multiple independent execution pipelines and multiple internal buses, thus providing a high degree of parallelism. The MC68040 also provides low latency bus accesses which lead to the reduction of cache miss penalties.

There have been extensions to both the user programming model and supervisor programming models of the MC68040. This is primarily a result of having an on-chip floating point unit. The user programming model also offers eight 80-bit floating point data registers (FP0-FP7), a 16-bit floating point control register (FPCR), a 32-bit floating point status register (FPSR), and a 32-bit floating point instruction access register (FPIAR). The floating point data registers are analogous to the data registers, D0-D7. The FPCR contains the enable byte which enables or disables the floating point exceptions and the mode byte that sets the user-selectable rounding and precision modes. The FPSR contains a condition code byte, a quotient byte, a floating point exception status byte, and a floating point accrued exception byte. The FPIAR contains the logical address of the instruction that can generate an exception trap. The additions to the supervisor programming model are the two additional transparent translation registers. The MC68030 offered a TT0 and TT1. The MC68040 offers data transparent translation register 0 (DDT0), data transparent translation register 1 (DDT1), instruction transparent translation register 0 (ITT0), and instruction transparent translation register 1 (ITT1). These four independent translation registers pertain to the memory management programming model which supplements the supervisor model. These registers are used in defining blocks of logical address space.

The MC68040 also retains the same 18 addressing modes as the MC68030, but it supports more data types. In addition to support the seven data types supported by its predecessor, the MC68040 also supports 16-Byte, single precision real, double precision real, and extended double precision real. The 16-Byte data format is a new data format used when moving a block of 16 bytes of data using the new instruction, MOVE16. The other new data formats are associated with using the floating point unit.

Unlike the MC68020 or MC68030, the MC68040 does not provide dynamic bus sizing so the MC68150 dynamic bus sizer is needed to communicate with the different bus size peripherals.

Although there are some differences in the signal groups between the MC68040 and its predecessors, it still maintains the 32 bit address bus, 32-bit data bus, bus arbitration signals, and clock signals. The MC68040 offers a variety of new signal groups which are the transfer attributes, bus transfer control, bus snooping control, and the test groups. The transfer attribute signals provide informational about the transfer such as the transfer type, transfer modifier, transfer line number, user programmable attributes, direction, transfer size, locking mode, and cache mode field. The transfer type indicates the type of transfer such as normal, MOVE 16, alternate function code, or acknowledge. The transfer modifier provides supplemental information about the access. The transfer line number is an indicator of which cache line is being pushed or loaded by the current line transfer. The user programmable attributes are user defined signals which are controlled by the user attributes in the address translation entry. The direction simply identifies the transfer as either a read or write. The transfer size pertains to the size of the data transfer. The locking mode indicates that a bus transfer is part of a read-modify-write cycle and should

not be interrupted. The cache mode field signifies that the current bus transfer will not be cached. The bus transfer control signals are used in controlling the bus transfer by providing indicators such as the start of a transfer, a transfer is in progress, a transfer acknowledge, and an error condition occurred during the transaction. The bus transfer signals are also used to inhibit any read data from being loaded into the instruction or data caches and to indicate that a slave processor cannot handle burst mode accesses. The MC68040 offers bus snooping in order to maintain cache coherency. The bus snooping control signals specify the snoop operation that is to be performed for an alternate bus master transfer and will inhibit memory devices from responding to an alternate bus master access during snooping operations in order to ensure that possibly stale data is not accessed. The test signal group on the MC68040 provides user-accessible test logic circuitry. This test access port is fully compatible with the IEEE1149.1 Standard Test Access Port and Boundary Scan Architecture.

## **68060**

Although the MC68060 is not yet available, preliminary information concerning its structure was obtained. The MC68060 offers increases in performance. Such increases are achieved by offering a high degree of parallelism, multiple execution units, and multiple internal buses.

The MC68060 retains the features offered by the MC68040 which include a floating point unit, two independent MMUs and two caches for instructions and for data. The caches have been increased to 8k.



The MC68060 still maintains object compatibility with its predecessors. It supports the same amount of addressing modes and data bytes as the MC68040. It also appears that the input and output signals are same as in the MC68040.

An enhancement to the MC68040 is the power management. The MC68060 has been developed to have low power consumption. When it is not use, it will not draw power.

## **VME BUS**

The VMEbus design was developed by Motorola in the early 1980's for the Eurocard based on their Versabus. The VMEbus was initially called the Versa Module Europe bus, but it is better known as the VMEbus.

NASA chose this bus because of its flexibility and the way it complements the 68000 family. The VMEbus can operate with a data width of 8,16, or 32 bits and with a 24 or 32 bit address bus.

The specifications concerning the VMEbus can be subdivided into the specifications concerning the physical structure and the functional structure. The specifications associated with the physical structure address the mechanical and electrical considerations. The specifications addressing the functional structure subdivide the VMEbus into four subbuses and describe the functionality of each of these four subbuses.

## VMEbus Physical Structure

The physical structure specifications are broken down into mechanical and electrical specifications. The mechanical specifications provide guidelines on the dimensions of the VME chassis and its cards and provisions that the backplane must make. The electrical specifications were developed to ensure proper timing and minimize noise.

The mechanical specifications provide recommendations on the physical dimensions, connectors, and backplane issues. The VME chassis can support either single height cards, double height cards, or a mixture of the two. The VME cards are housed in a chassis that is 482.6 mm wide but may be either 132.5 mm or 265.9 mm high. Depending on the height of the chassis, it can support either support single height cards (100mm x 160mm) or double height cards (233.35mm x 160mm). The chassis may support up to 21 slots, thereby, having a width of 482.6 mm wide. One card may occupy more than one slot. The recommended thickness of a card is 1.6 mm.

The VMEbus offers two connectors, P1 and P2. P1 provides all the functions that are necessary in implementing a basic VMEbus. P2 offers expansion facilities such as increasing the address bus from 24 to 32 bits and increasing the data bits from 24 to 32. The connectors on the VME cards are labeled P1 and P2, but the connectors on the backplane are called J1 and J2. The VME backplane may be either configured as a single backplane where only P1 is used or as a double backplane where both P1 and P2 are utilized. The P1 and P2 connectors consist of three rows of 32 pins and conform to the DIN 41612 standard.

The provisions associated with that the backplane include busing of signals, jumpering, and using terminators. Except for the daisy chained signals, the J1 connector must bus all signals. The backplane must provide for jumpering of the interrupt acknowledge and bus grant daisy chains when boards are present. The backplane must either have built-in terminators or must provide some way of connecting plug-on terminator boards for terminating signal lines. Another issue associated with the backplane is that signal traces may not exceed 508 mm. There are also limits on the traces from the connectors to the on-board circuitry.

Since the backplane provides the pathway for the transmission of data between VME boards, electrical specifications were developed to ensure proper timing of signals on the backplane and the minimization of any noise or crosstalk of these signals. The electrical specifications provide guidelines concerning voltage ratings, contact resistance, and insulation resistance. The electrical specifications imposes certain regulations concerned with termination, power conductors, and ground connections. For instance, for lines that require termination a means of terminating lines at both ends is required. Power conductors must be provided for the distribution of +5V, +5V STDBY, +12V, -12V to all specified power pins. Another rule is that ground conductors must be provided to all specified ground pins. In order to minimize electrical problems, electrical specifications provide driving and loading rules. Boards must not drive any backplane signal to states beyond the highest voltage or lowest steady state voltage. Boards are required to use drivers and receivers. Three-state, open collectors, and totem pole drivers are considered possible drivers. The detection of low and high logic beyond the threshold must be guaranteed by the receivers.

## **VMEbus Functional Specifications**

The VMEbus can be logically divided into four subbuses which are: the Data Transfer Bus, Data Transfer Arbitration Bus, Priority Interrupt Bus, and Utilities Bus.

The Data Transfer Bus (DTB) provides the data and address pathways and associated control signals which make it possible to transfer data from a master to a slave. A bus master is a functional module which is capable of initiating bus transfers. A slave is a functional module which responds to a data transfer. The DTB may be considered an extension of the asynchronous bus of the 68000 family of processors. The transfer of data to a slave requires that a module obtain control of the bus if it is not already a bus master. The control of the bus is acquired via a bus requester module. The DTB also provides a special feature which is the 6-bit address modifier bits (AM0-AM5). These bits allow the master to provide six bits of additional information to the slave. The address modifier bits may be possibly used for system partitioning, memory map manipulation, privileged access, and address range determination.

The Data Transfer Arbitration provides the means of transferring control of the Data Transfer Bus between modules that are each capable of being bus master. The Data Transfer Arbitration facilities are only required by systems which have more than one module capable of being bus master. Such modules usually have a processor. The two modules that partake in the bus arbitration process are the bus arbiter and the bus requester. The bus arbiter module, which resides in slot 1, is a functional module that receives from the other modules requests to obtain the Data Transfer Bus. The bus arbiter prioritizes such requests and grants the bus to the appropriate requester. The bus requester module resides in every card that is capable of being bus master.

The arbitration bus consists of fourteen lines which are divided into two groups. One group refers to lines which are driven by the bus requester module. The other group is characterized by the lines that are driven by the arbiter. The lines driven by the requester are the bus request 0-3, bus grant out, and bus busy. The bus request 0-3 lines are used by the requester to gain access to the Data Transfer Bus. The bus grant out 0-3 lines are used by the requester to pass down the bus grant in signal sent by the arbiter. The bus busy signal indicates that the bus is currently in use. The lines that are driven by the arbiter are the bus clear and the bus grant in 0-3. The bus clear is used to inform the current bus master that the bus is needed. The bus grant in 0-3 lines are used by the arbiter to indicate that bus has been granted at the corresponding level.

The bus arbiter must implement a strategy for prioritizing the requests for the bus. Possible strategies include the round robin select method, the prioritized option, and the single level method. The round robin select option assigns priority to the masters on a rotating basis. The arbiter cycles through the four levels of the bus request (bus request 0-3) testing for a bus request. If the bus level does not need to use the bus, the arbiter will go to the next level. The levels are tested in the following order: bus request 0, 3, 2, and 1. The round robin select option provides a fair method of arbitration since all levels are checked and no level is ever excluded. The prioritized option assigns a priority level to each level with bus request 3 having the highest priority and bus request 0 having the lowest. This method is not a fair method of arbitration because a low priority level may not be serviced if a higher priority is always requesting the bus. The single level strategy offers minimal bus arbitration by offering only one bus request level, namely bus request 3. The priority of the modules is based on the location of the requesting module. The closer the module is to the arbiter the higher the priority. This is a result of daisy chaining. A

daisy chained line is a unidirectional line where a signal propagates from one module to another until the requesting module obtains the bus grant.

The Priority Interrupt Bus enables the modules to request service from a Data Transfer Bus master. The Priority Interrupt Bus and its associated modules extend the interrupt-handling capabilities of the 68000 family of processors. The Priority Interrupt Bus consists of ten bus lines. These ten lines are the interrupt request 1-7, interrupt acknowledge, interrupt acknowledge input, and interrupt acknowledge output. The modules associated with the interrupt priority bus are the interrupt requester and the interrupt handler. The interrupt requester is also known as the interrupter and is a functional module which is capable of requesting service from a master. The interrupter accomplishes the following: generates the interrupt, supplies the status ID when the interrupt request is acknowledged, and passes the interrupt acknowledge daisy chain signal if it has not requested service. The interrupt handler is a functional module which detects interrupt requests and initiates the appropriate responses to such requests. The interrupt handler performs the following: prioritizes the interrupt requests, acknowledges the interrupt, reads the status ID from the interrupter being acknowledged, and initiates the appropriate interrupt service routine.

The Utilities bus consists of miscellaneous signals. These signals are the system clock, system reset, system fail, and the AC power failure signal. The system clock, SYSCLK, is a master timing signal and serves as the source of timing for all modules on the VMEbus. The system reset, SYSRESET, is driven by the power monitor module or a manual reset switch. SYSRESET will place the processor in a known state on power-up or a manual reset. The system fail signal, SYSFAIL, indicates when the system has failed. The VMEbus standard does not define what constitutes a system failure. The AC power

failure signal, ACFAIL, indicates that the AC power supply has failed and is no longer operating within the specified operating range.

## **CHAPTER 4**

### **ACTUAL MODIFICATIONS**

The PCOS kernel was originally written in assembly with three versions currently being maintained. Each version corresponds to the microprocessors used in the communications system, so the 68010, 68020, 68030 versions exist and must be maintained. These three versions have been combined into one C language version of the kernel. This will easily support any possible upgrades and will remove any further proliferation. The modifications not only include translating the kernel into C but will also include fixing any problem areas. When software bugs are found in the kernel, problem reports are generated. The detection of a software bug does not mean that a correction will be immediately implemented. The impact due to the presence of the bug is analyzed versus any schedule impacts created by fixing the error. Problem reports were also reviewed and corrections were implemented in the new kernel.

The discussion of the kernel has been divided into sections based on the functionality. These sections are initialization, scheduling, system service calls, memory management, I/O services, exceptions, and interprocess communications.



## INITIALIZATION

Although the initialization is primarily handled by the PCOS debugger, the kernel must also perform some initialization. This initialization includes allocating and establishing certain data structures and tables such as the vector table and service tables, searching for modules, establishing linked lists, and starting the Kernel Extension Module process and the initial process.

The initialization routine, `init`, will request memory and will initialize this memory according to the requirements provided in the `init` table module. Since the initialization routine relies heavily on the services provided by the debugger such as memory allocation, the PCOS debugger will pass to this routine a structure containing certain necessary parameters. Such elements include the MMU minimum block size, pointer to the `init` table, and pointers to the debugger services and debugger handlers. These elements are saved in the direct page which includes parameters that the kernel will use extensively. The `init` routine will request memory according to the `init` table. The pointers to such memory requests are saved in the direct page. For instance, the `init` table module provides the number of mailboxes or queues that are needed. `Init` will request memory in order to satisfy the queue size requirements listed in the `init` table module. If any errors occur in the allocation of the memory, an error message will be displayed. The `init` routine will request memory for the following: a 2k exit stack, dispatch table, module directory, process descriptors, path descriptors, device table, mailboxes, semaphores, and queue structures.

The init routine will also setup the vector table. This table is established using the existing vector values and the contents of the kernel vector image table. If the kernel vector image table indicates that the existing vector should be left unaltered or if the existing vector table does not point to the debugger's unimplemented exception handler, the vector table is not modified. Otherwise, vector table location is updated. The bus address error, illegal instruction and trace exception handlers are also setup. The system and user service tables are also established.

A search for all modules is also performed by the init routine. Each module contains a sync word. This sync word is not a valid opcode combination. A search of the memory is done for such sync words. In the case of the Motorola 68000 family, the sync word is 4AFC. Once the sync word is encountered, the module size is obtained and the module is validated. Such validation includes using the module size and performing a checksum based on the memory space indicated by the module size.

The init routine will also establish the linked lists for the process descriptors and the path descriptors. The number of process and path descriptors is obtained from the init table. Using this number and the size of the descriptor, the init routine will save the pointer to the next descriptor in the current descriptor. The pointer to the first and last descriptor is saved in the direct page.

While the init routine is performing such initialization and prior to the start of any process, the kernel marks a boot active flag. The init routine will also save the address of the tick routine in the direct page while this flag is active. This flag is marked inactive prior to activating any process.

The kernel searches for the existence of the Kernel Extension Module process. If such a module exists, a process descriptor is obtained from the free list and is established for this process. This process is then forked. The Kernel Extension Module can be used for certain customization of the kernel for certain applications. After the Kernel Extension Module is forked or if it does not exist, the initial process is forked. The initial process is identified in the init table.

## **SCHEDULING**

The scheduling of the process is performed using the tick routine. The tick routine processes the tick interrupt, checks the active process queue, and dispatches the next process ready for execution. Although there may be many runnable processes, only one is executed. The runnable processes are on the active process queue.

The real time clock interrupt is referred to as a tick. The tick routine receives this interrupt and processes the interrupt. The processing of such interrupt includes decrementing the sleep tick count at the head of the queue of the sleeping processes, the timer tick count, and the tick slice for the current process and updating the counters pertaining to dates.

When the tick count at the ahead of the sleep queue reaches zero, the process is moved from the sleep queue to the active queue. The way the sleep queue is designed is that processes are sorted by their sleep tick count. The sleep tick count refers to how many ticks is the process to be asleep for. The sleep queue maintains a tick count at the head of

the queue and a delta of ticks for each process in the queue. This delta is the difference between the sleep tick count for each process and the tick count at the head of the sleep queue. By decrementing the sleep tick count at the head of the queue, the tick routine does not need to decrement each tick count for each process in the sleep queue, thus decreasing the processing time. The tick routine not only moves the process at the head of the sleep queue to the active queue when the tick count is zero, but it will also move any processes having a delta of zero into the active queue.

Once the processing of the sleep queue is complete, the tick routine checks to see if any timers are enabled. If the system has enabled any timers, the tick count for the timers is updated. A timer routine is called upon to do any processing pertaining to a timeout.

After the timers have been processed, the tick routine updates the counters related to the system date and time. The counters include the counts of ticks per day, seconds per days, Julian date, day, month, and year. Using the number of ticks per second and the number of ticks in days, the tick routine will check if there has been a day rollover. Once the day rollover occurs the Julian date and day counters are incremented. The month rollover is then checked, and the month counter is updated accordingly. The year counter is updated if a year rollover has occurred.

Once the counters have been updated, the tick routine checks whether the current process is operating in the system or user state. If the current process is in system state, the count of system ticks is updated; otherwise, the count of user ticks is incremented.

Since each process is awarded a certain time slice, the tick routine needs to check if the time slice for the current process has not expired. If the current process still has time

remaining, it does not relinquish the processor and continues its execution. If there is no more time, the tick routine then verifies that the current process is not in the locked mode. The lock mode refers to a process locking out other processes from obtaining control of the processor for a period of time. If the current process is indeed in the lock mode, the tick routine will determine if lock timer has expired. Once the lock expires for a locked process or the timeslice for the current non-locked process expires, the process is placed in the active queue in accordance with its age and priority to wait for the processor. The process at the top of the active queue is then given control of the processor.

One of the responsibilities of the tick routine is to insert a process into the active queue and to sort the active queue. Processes are placed into the queue and sorted based on their age. The age of the new active process is equated to its priority. The age of each process already present in the active queue is incremented by one. The age of the new process is compared with the age of each process in the active queue. The new process is inserted into the queue based on its age. The higher the age the closer to head of the of the queue a process is inserted.

The tick routine is not only responsible for scheduling processes but also dispatching the process. The process at the head of the active queue is granted the processor and can begin executing. The dispatching of a process involves setting up the environment under which the process was executing prior to a task switch. This means that memory segments may have to be activated and registers restored. Once a process is dispatched, the active queue is then adjusted to reflect the removal of a process from the head of the queue.

## SYSTEM SERVICE CALLS

When a PCOS system service request is made, a trap is issued. Trap 0 has been reserved for PCOS system calls. Since the syscall function is dedicated to responding to the trap 0, all PCOS system calls are handled by the syscall function. With the issuing of the trap 0, a system service request code is also passed. This system service request code identifies the system call that is being issued. This code serves as an index into the table of system service calls. The syscall function will verify that the code points to an existing system service. In addition to validating the system service request code, it must verify that the user process can access it. Some service request calls have restricted access meaning that only processes in the supervisor mode can issue them. If a user process attempts to issue a restricted system call, the syscall function will deny the request. Before calling the function which handles the specified service request, the syscall is responsible for the saving the context of the user process and setting up a structure which contains all the necessary arguments that the service function will need. When the service function returns to the syscall function, the syscall will then obtain the return arguments from the service function and pass them back to the user process.

The system service calls are facilities which include creating and terminating a process, changing the state of the process, performing memory management services, handling exceptions and I/O, and enabling communications between processes.

The fork system call is used in the creation of new process. The new process is said to be the child of the current process. Since a new process will be created, the kernel must allocate a process descriptor. Once the process descriptor has been allocated, it must be

initialized accordingly. Much of the process descriptor is initialized according to the values in the parent's process descriptor. For instance, the child will inherit the default I/O paths. If the child is to inherit the other I/O paths that the parent may have, it must be specified. The initialization of the new process descriptor also includes obtaining the incarnation number of the process and the priority. The incarnation number is obtained by incrementing the incarnation number currently maintained in the direct page. The direct page maintains the largest incarnation number. The priority is either specified at the creation or the default priority obtained from the direct page is used. Not only is the process descriptor belonging to the child initialized but the parent's process descriptor is modified to show that it has a new child. This new child is marked as the youngest child if more than child process exists. A process descriptor belonging to any sibling must also be modified to show the relationship with the new process. Once the necessary process descriptors are updated and initialized, the new process is placed on the active queue.

The system services pertaining to the allocation and deallocation of the process and path descriptors are found in the pds routine. The allprc function will allocate a process descriptor from the free process descriptor list. Its counterpart, retrpc, will simply return the process descriptor to the free process descriptor list. The allpath function obtains a path descriptor from the free path descriptor list while the retrpath will return the path descriptor to the list.

The exit system call is used to terminate a process. This system call will check if the parent process is in the wait state waiting for the child process to terminate. The parent process that had been waiting is then placed in the active queue. The process descriptor belonging to the parent process is altered to indicate that the child process no longer exists. If the process to be terminated has any siblings, the process descriptors associated

with the sibling processes are also modified to reflect the termination of the process. If the process to be terminated has any child processes, these child processes are classified as orphans. Part of termination of the process includes certain clean up activities such as closing all opened paths, unlinking from any modules, and releasing any semaphores it may own. Once these actions are complete, all memory pertaining to the process is returned to the pool of free memory.

The wait system call is responsible for placing a process in the waiting process queue. Only processes which have child processes are placed on the waiting process queue. The processes which are on the waiting process queue remain on the queue until a child process terminates. Upon termination of the child process, the parent process is removed from the waiting process queue.

The sleep system call is used for placing a process in the sleeping process queue for a specified length of time. If the specified length of time is zero, this means that the process is to be sleeping for indefinite period.. PCOS maintains two different sleeping process queues, one for indefinite sleep and the other is for timed sleep. Processes with a specified sleep time of zero are then placed on the indefinite sleep process queue while process with a non-zero sleep time are placed in the timed sleeping process queue. Process are placed in the timed sleep process queue in increasing order of sleep time. At the head of the queue is the tick count of the process with the smallest sleep cycle. The processes in the queue are sorted based on the difference between the process's sleep cycle and the smallest sleep cycle. By having only to maintain the count at the head of the queue, processing of this queue is decreased because PCOS does not have to traverse the queue decreasing the tick counts of each sleeping process.



## MEMORY MANAGEMENT

The memory management functions are broken into two categories: memory allocation and module related. The memory allocation functions service requests for memory and return the memory to the free pool once the memory is no longer needed. Although the kernel provides memory allocation services, the work of obtaining the memory and returning the memory is primarily done by the debugger. The kernel provides the interfaces and services the requests but relies on the debugger services to do the work. The module related functions deal with PCOS memory modules. The module related functions maintain the module directory and provide facilities for linking, unlinking, loading, unloading and validating memory modules. These module related functions are dependent on the memory module format.

The memory allocation functions are `srqmem`, `srtmem`, and `setmem`. These functions are part of the `mem` routine. The `srqmem` function handles requests for memory while the `srtmem` returns the memory to the free pool. The `srqmem` relies on the debugger service, `extract`, to obtain memory. Before using the `extract` debugger service, `srqmem` first checks to see the state of the system. If the state of the system reflects the supervisor mode, the `extract` function is called. If the mode indicates that a user process made the request for more memory, the `srqmem` function searches the memory block table for unused entries. The process descriptor will contain a pointer to the memory block table. When a process is created, PCOS will automatically allocate a contiguous block of memory for stack and data usage. The pointer to this primary memory block and the size of this block are stored in the memory block table. PCOS allows a process to have up to 32 discontinuous blocks of memory; each block of memory consists of contiguous

memory. Pointers to these blocks and their corresponding sizes are contained in the memory block table. If the search of the memory block pointers shows that the process does not have 32 blocks already allocated, the extract function is called to allocate another block to the process. The pointer to the newly allocated block and its size are then stored in the table. If the table is already full or the debugger reports errors during the extract, the user process is notified of such problems.

The `srtmem` function uses the debugger service, `liberate`, in order to return the memory to the free memory pool. This function also checks the state of the system. If the system is in the supervisor mode, the debugger service, `liberate`, is automatically called. If a user process made to request to de-allocate the memory, the memory block table is searched to verify that the pointer to the block to be allocated actually exists. If the pointer exists in the table, the `liberate` function is called upon to de-allocate the block. Once the `liberate` function is done, the pointer and the block size are deleted from the memory block table. If any errors occur during the deallocation or the pointer is not found in the table, the user process is informed.

The `setmem` function is responsible for setting the memory size of the primary block of memory. The size requested is compared with the current block size. If the size requested is larger than the current block size, a contiguous block of memory is allocated. The pointer to this block and its size is stored at the top of the memory block table indicating that this new block is the primary block. If the size requested is less than or equal to the current block size, the kernel then checks to make the size requested is sufficient for the stack area. If the stack area is safe, the debugger service, `liberate`, is used to deallocate the remainder of the memory. This remainder is the difference between the original size and the size requested.

The module related functions are contained in the crc routine and module routine. The crc routine is responsible for validating a module, performing a CRC calculation, and generating a module header parity and CRC. The validation of a module is accomplished by the vmod function. The vmod function receives the address of a module and returns a pointer to the module directory entry. The vmod function compares the CRC contained within the module with the CRC that is generated using the module contents. If the module CRC is equal to the generated CRC, the module directory is searched. If the module is not found in the module directory, memory is requested and the module is incorporated into the module directory. The link count for this module is set to one. If a module is found in the module directory with the same name, the two modules are compared in terms of their edition and revision numbers to determine which of the two is to be used. The crcgen function will generate a CRC value for a specified number of bytes beginning at a given location. The setcrc function is responsible for generating a module header parity and CRC value. This function just deals with the module header and not the module body.

The module routine encompasses other module based facilities such as linking to a memory module, unlinking from a memory module, loading and unloading a data module, building a data module, and searching for a memory module. The link function is responsible for linking to a data module. This function must first find the module in the module directory. Once the module is found in the module directory, the attributes are checked to determine if the module is re-entrant. If the module is not re-entrant, the process will not be able to link to it. The link count will be incremented for re-entrant modules. Another reason for checking the attributes is to obtain the write and read permissions associated with the module. If the module has read only permission, the

kernel will not allow the process to write to the module. The kernel will mark the memory area pertaining to the module as write protected. The unlink function is the counterpart to the link function. The unlink function will also search the module directory searching for the specified module. Once the module is found, the link count is decremented signifying that one less process is using the data module. The load function will place a file into memory and record its presence in the module directory. The link count for the newly loaded module is initialized to zero which indicates that no process is yet utilizing this module. The unload function is the counterpart of the load function. The unload module will remove a module from memory and the module directory only when the link count is zero or -1 for sticky modules. The datmod function is used to build a data module. The address of the data module name, the size of the module body, the attributes, and the access permissions must all be specified when creating a data module. This function will extract enough memory for the body, header, and CRC. The module header is then initialized with the module id values, attributes, size, and permissions. The function setcrc is then called to obtain the module header parity and CRC. The CRC for the entire module must also be calculated. Once the module has been initialized, the module is then validated. The findmod function is responsible for traversing the module directory and searching for the specified module.

## **I/O SERVICES**

PCOS provides system-wide hardware independent I/O services. The kernel receives all I/O service requests, does some processing such as allocating any necessary data

structures for the I/O path, and calls the corresponding file managers and device drivers to continue the processing. The kernel routine that handles all I/O service requests is the `iocall` routine. This `iocall` routine consists of the various I/O system services such as: `attach`, `chgdir`, `close`, `create`, `delete`, `detach`, `discon`, `dup`, `getstt`, `makdir`, `open`, `read`, `readln`, `setstt`, `write`, and `writeln`.

The `attach` service request is used to either attach a new device to the system or verify that a device is already attached. The module directory is searched to determine if a device descriptor exists with the given device's name string. If the corresponding descriptor module is found in the module directory and the device is not already attached, PCOS will then link to the file manager and device driver as defined by the descriptor and incorporate the address into a new device table entry. The kernel then allocates any storage that the device driver may need and calls the driver initialization routine. If the device has already been attached, the count pertaining to the usage of the device is incremented. The `attach` system call is not required in order to perform routine I/O. This system service just prepares the device for subsequent use by any process. Since most devices are automatically installed, this I/O system service is generally used when devices are being installed dynamically or to verify the existence of a device.

The `chgdir` service request is used to change the default data or execution path of a process. When a path is given that does not include the device specification, the appropriate default path is utilized in any subsequent I/O calls. The default path specified in the `chgdir` is then appended to the path specified in the I/O call.

The `close` service request is responsible for terminating the I/O path associated with the given path number. Once the path to a file or device is closed, I/O can no longer be

performed to the file or device without issuing an open or create service request. Once a path is closed, the data storage is deallocated. The EXIT system service request generally will close all open paths.

The create service request is used when creating a new file on multiple file devices such as printer or terminals. PCOS returns a path number which is used to identify the file in subsequent I/O service requests. PCOS does not allocate any data storage upon creation. The allocation of storage is done automatically by the write service request or explicitly by the setstt service call.

The delete service request is used for deleting the file as specified by the pathlist. PCOS first checks the attributes to see if it has write permission before deleting the file. Any attempts to delete devices which do not have write permission result in an error.

The detach service request handles the removal of a device from the system device table as long as the device is not being used by any other processes. The termination routine of the device driver is called. The storage that has been assigned to the driver is then deallocated. PCOS will then unlink from the associated file manager and device driver. The detach service request is used to unattach devices that were attached using the attach service request.

The discon service request is used for displaying information on a console. This service request is to be used only in conjunction with SCF devices. The operation performed by this service request depends on the function code that is passed to the service call. The function codes represent actions which govern the display. For instance, the function

codes govern the movement of the cursor, the physical representation of the cursor, video mode, and tabs.

The dup service request obtains a synonymous path number for the same file or device associated with the given existing path. With this service call, the use count associated with a path descriptor is incremented and the synonymous path number is returned. The path descriptor is not copied. The dup essentially duplicates an existing path. This service request is used with the redirection of I/O.

The getstt service call provides the status of a file or device associated with a given path number. In addition to the path number, this service call requires another input which is a function code. These functions codes are either predefined or can be user defined. The predefined codes refer to reading the option section in the path descriptor, testing for the readiness of the data on an SCF device, and testing for an end of file. The operation of this service call depends on the given function code, the device driver and file manager associated with the given path.

The open service request opens a path to a file or device as specified by the given pathlist. The open service call returns a path number which is to be used in subsequent service requests to identify the file. The service call also requires an access mode parameter. This access mode parameter signifies whether subsequent read and/or write operations are permitted. A restriction concerning the opening of devices is that devices cannot be opened simultaneously. In fact, devices have an attribute that specifies whether they are sharable or not.

The read service call is used for reading a specified number of bytes from a given path number. The path must have been previously opened with a read access mode. The data that is returned using the service call is exactly as it was read from the file or device without additional processing or editing such as backspaces or end of file markers. The number of bytes requested is read from the file or device unless an end of file occurs, an end of record occurs, or an error condition occurs. The readln service call differs from the read service call in that the readln service request reads a text line from a file or device until a carriage return character is encountered. Like the read service call, the readln call will read until the number of bytes specified are read.

The setstt service call is used for setting the status of file or device. This system call can be used to handle device parameters that are not uniform on all devices, are highly hardware dependent, or need to be changed by the user. The system call requires two inputs: a path number and a function code. The operation of this call depends on the function code, the file manager and device driver associated with the given path. The function codes can be either predefined or user defined. The predefined codes are for writing to the path descriptor options sections and for issuing form feeds in the SCF devices. This service call is typically used for setting the device operating parameters such as echo or auto line feed.

The write service request will write the number of bytes specified to a file or device associated with the given path number. The path must have been opened with the write access mode. The data is written to the file or device without processing or editing. The writeln service call will also attempt to write to the file or device specified by the path number the specified number of bytes, but it will stop writing when a carriage return character is encountered. With the writeln service call, line editing is also activated for



character-oriented devices such as terminals and printers. Line editing refers to auto line feed or null padding at the end of the line.

## **EXCEPTIONS**

The exceptions routine contains all functions that deal with exception handling. The exceptions routine consists of the following functions: setrap, setvec, setue, unimpl, syscrash, buserr, adrserr, illinst, zerodiv, chkbnds, overflow, privviol, trace, in1010, in1111, exsetup, userexset, trapsetup, and the trap functions. These functions are either the exceptions handlers or the functions which install the exception handlers. Some of the exception handlers included in the exceptions routine are very specific to exceptions associated with the Motorola family of processors. These are the privviol, chkbnds, in1010, in1111, and to a certain extent the trap functions.

The setrap function is responsible for installing or removing a user trap handler. This function first verifies if the exception number is a valid one. Once the exception number is validated, the pointers to exception handler and to the data storage are placed in the tables where traps are listed. If the request is to remove a user trap handler, the pointer to both the exception handler and the data storage are removed from the table.

The setvec function places the given exception vector in the exception vector table. The specified exception vector is validated to ascertain that it lies within the range of the valid vector numbers.

The `setue` function is responsible for establishing the unimplemented exception handler in the vector jump table. The `setue` function will verify that the pointer to the jump table is non-zero. If the pointer to the jump area is zero, the jump area is then cleared. Otherwise, the `setue` function verifies that the vector has not already been installed. If the vector has not been installed, the `setue` sets up the pointers to the exception handling routine in the jump area. The exception handling routine which deals with the unimplemented exceptions is the `unimpl` function. This handler is invoked for every unimplemented exception. Using the `setue` function, it determines whether a user handler has been installed, and if so, the `unimpl` will call the user installed handler. If a user handler has not been installed, this function simply increments a count of unimplemented exceptions.

The `syscrash` function is called when an exception occurs and no process is running. This function simply prints an error message and calls on the system debugger to display the contents of the registers as they were when the error occurred. The following functions all utilize the `syscrash` function if there is no process running: `buserr`, `adsrerr`, `illinst`, `zerodiv`, `chkbnds`, `overflow`, `privviol`, `trace`, `in1010`, `in1111`, and the `trap` functions. All these functions first check to see if a current process was running when the exception occurred. If no process was running, these functions call the `syscrash` function. If a process is running, the functions then check if the current process has a user exception handling routine that needs to be installed. The `buserr` and `adsrerr` functions rely on the `exsetup` function to install the user handling routine. The `exsetup` function will copy the entire stack frame to the user stack. Once the `exsetup` is finished copying the stack information, the user exception handler is called. The other exceptions, `illinst`, `zerodiv`, `chkbnds`, `overflow`, `privviol`, `trace`, `in1010`, and `in1111`, rely on `userexset` for establishing the user handling of such errors. The `userexset` function copies the program counter,

format code, and vector offset from the system stack to the user stack. Once this information is copied to the user stack, the user exception handler is called. If the current process does not have any user exception handling routine to handle any of these exceptions, the exit system call is issued. The trap functions rely on trapsetup for the installation of any user trap handler. The trapsetup function will copy function codes, format and vector offset from the stack to the user's stack. The trapsetup function will then call the user trap handler. If a user trap handler does not exist, the exist system call is issued.

The buserr function handles bus errors while the adrserr function deals with exceptions resulting from using an invalid address. The illinst exception handler is called when the system encounters an illegal instruction. The zerodiv function is designed to handles any attempts concerning a division by zero. The chkbnbs function pertains to checking the boundaries. The ovrflow function handles any exceptions resulting from overflow problems. The privviol function addresses any exception resulting from the violation of a privileged instruction. The in1010 and in1111 functions pertain to the emulation modes. The trace function deals with all facilities concerning the tracing of software. The trap functions are essentially software interrupts. For instance, they can be used for issuing a system service request, invoking the I/O or the math functions. These three traps have been defined by PCOS, but Motorola family offers up to sixteen traps, thereby, the user may define the other thirteen traps as needed.

## TIME BASED SERVICES

The time based functions are broken down into two classes: timers and system time functions. The timer function process any timeouts while the system time functions provide the system time and date facilities.

PCOS provides facilities by which processes can establish timers and remove timers. The `setimer` routine creates and initializes timer; the `clrtimer` routine removes any timers. The `setimer` searches for a free slot in the timer group table and adds a timer to the timer group and sorts the timers according to expiration time. The `clrtimer` routine searches the timer group table and removes the timer from the table once the timer is found. The processing of any timeouts is done by the `timeout` routine. This routine is called upon by the `tick` routine when then the timer tick count is zero. When the timer tick count, the `timeout` routine checks through each timer group to determine how many timers have expired. When a timer expires, an event is sent to a process to indicate that a timeout has occurred. Timers can be setup such that an event is sent to itself or another process. Once the event is sent, the `timeout` checks to see if the timer is a repetitive. If the timer is repetitive, the timer tick count is reinitialized and the timer is still part of the timer group.

The `time` routine provides the facilities that either set or return the system time and date. The `setime` function sets the system time and date. The function, `setime`, checks to see if a user process is requesting to change the system time and date. User processes can change the system time and date, but only processes in the supervisor mode can change the ticks per timeslice. Before changing the system date, the `setime` function verifies that the given date and time are valid. The `setime` function will then link to the clock module and execute the clock's initialization routine. After completion of the clock's initialization, the

setime function updates all date and time counters that are updated by the tick routine. The Gregorian date is converted to the Julian date so both the Julian and Gregorian dates are maintained. The rettime function returns the system time and date. The rettime function will return either the Gregorian or Julian date depending on what format the user requested. In addition to the system date, the retime function will return the day of week, ticks per second, the current tick count, and the ticks that have elapsed since midnight.

## **INTERPROCESS COMMUNICATIONS**

The interprocess communications services offered by PCOS include signals, mailboxes, queues, events, and semaphores. Since there are a variety of services provided in terms of interprocess communications, a variety of routines were developed in order to provide such facilities. These routines are the comm routine, queue routine, the events routine, and the semaphore routine. Each of these routines contains functions which provide the necessary facilities.

The functions pertaining to signals and mailboxes are found in the comm routine. The comm routine contains the following functions: send, post, and getmail. The send function is responsible for sending a signal to another process. The receiving process's id number and a signal code are the inputs to the send function. This function verifies that the given process id is a valid one and that the process is not dead. Once the process id has been validated, the kernel checks the current status of the process. If the process is in one of main inactive queues, the process is not awoken and the signal is stored. If the process is not in one of the main inactive queues, the process is then placed in the active

queue and the signal is stored and is to be processed by the signal handler associated with the receiving process.

The functions associated with the mailbox system are the post and getmail functions. The post function sends mail to a specified mailbox, and the getmail retrieves mail from a specified mailbox. Both of these functions have another input which signifies if the process is to wait. The wait option flag signifies to the post function whether or not to wait for the mail to be sent if the mailbox currently has mail. The getmail function uses the wait flag to determine if the process needs to wait for the mail to arrive if the mailbox is empty. The first thing that both functions do is verify that the specified mailbox is a valid one. The post function then checks to see if the mailbox is empty. If so, it proceeds to place the mail value into the mailbox location. If the mailbox contains mail, the wait option flag is used to determine whether to exit the post function or to place the process in the waiting to post queue. If the process is placed in the waiting to post queue, the process will give up its timeslice. After validating the mailbox, the getmail function verifies that the mail has arrived. If the mailbox contains the mail value, the getmail retrieves the mail. Otherwise, the getmail functions checks the option flag to determine what needs to be done. If the option flag indicates that the process is to wait until the mail is placed in the mailbox, the process is then placed in the wait to get mail queue and gives up its timeslice.

The queues routine contains the functions which deal with the queue system. These functions are enqueue, dequeue, qstat, and qclear. The enqueue function will place queue elements on a specified queue. Besides needing the queue number, other inputs to this enqueue function include an wait option flag, the element size, and the pointer to the queue element that is to be enqueued. The enqueue function first verifies that a valid queue

number was given. Once this verification is complete, the queue is checked to determine if there is room. If the queue has room for the queue element, the element is placed on the queue. If there is not sufficient space on the queue, the wait option flag is checked to see if the process wants to wait until there is room on the queue. The process is placed on the waiting to enqueue queue and will give up its timeslice if the option flag indicates that the process is to wait until there is room. The deque function retrieves an element from a specified queue. This function also verifies that the specified queue is a valid queue. The queue is checked to determine if there are any elements in the queue. If the queue is empty, the wait option flag is checked. If the process is to wait until the queue contains elements, the process is then placed in the waiting to dequeue queue and will give up its timeslice. If the queue is not empty, the queue elements are removed from the queue and placed in the specified buffer. The waiting to enqueue queue will be checked by the deque function once the deque function completes its dequeuing to see if any processes are waiting for the specified queue to free up. The deque function will awaken the process on the waiting to enqueue queue as space becomes available on the queue. The wait to dequeue queue is checked by the enqueue function once the enqueue function completes its enqueueing. The enqueue function will wake up the processes in the wait to dequeue queue. The qstat function provides the status of the specified queue. The status reflects how many elements are currently on the queue and the size of the queue elements. This function first checks if the given queue number is a valid one. Once the queue number has been validated, the status information is obtained and returned. The qclear function is responsible for flushing out a queue. This function validates the given number and verifies that the queue contains elements. This function will remove all the queue elements in the given queue and will wake up any process waiting to enqueue to the given the queue.

The events routine consists of the two functions that constitute the event system. These functions are the `sendevnt` and `getevnt` functions. The `sendevnt` function will send an event to a specified process. The `sendevnt` has four inputs: the destination process incarnation number, the destination process id number, the opcode, and the address. The opcode and the address inputs do not convey any information to the `sendevnt` function. These two inputs are only meaningful to the destination process. The `sendevnt` function does use the destination process id and incarnation number. In fact, the first thing, the function does, is verify that these two are valid inputs. The `sendevnt` function checks if the event queue for the destination process is full. If the event queue is not full, the opcode and address are then placed in the event queue. A count of the events in the event queue is maintained. Once the opcode and address are stored in the queue, the count of the events is incremented. The `sendevnt` function also checks if the destination process is currently waiting to receive an event. If the destination process is waiting for an event, the process is placed in the active queue. The `getevnt` function will retrieve an event from the event queue. The counter of events in the queue is checked to make that an event has been received. If the event count is zero, the option flag is then checked to determine if the process should wait for an event to arrive. If the option flag indicates that the process is to wait for an event, the process is then placed in the wait for an event queue and will give up its timeslice. If the count of events does show that an event has been received, the opcode and address are extracted from the event queue and the count of events is decremented.

The semaphore routine contains the `getsem` and `relsem` functions which manage the semaphores. Both of these functions will verify that the specified semaphore is a valid one. If the semaphore is available, the `getsem` function will obtain the semaphore and set the semaphore according to the specified value. If the semaphore is currently is use, the



getsem function uses the wait option flag to determine if the process is to wait until the semaphore is available. If the process is to wait until the semaphore is available, the process is placed on the wait for semaphore queue and will give up its timeslice. The relsem function releases the semaphore. This function verifies that the semaphore actually belongs to the current process. Once the semaphore is released, the relsem will check to see if any process was waiting to obtain the semaphore. If a process was in the wait for semaphore queue, the process will then be activated.

## **CHAPTER 5**

### **CONCLUSIONS**

When converting the PCOS kernel from the various assembly language versions into a C language based kernel, several issues had to be addressed and analyzed. These issues include selecting the compiler, rewriting portions of the PCOS that are not considered part of the kernel, correcting the current problems within the kernel, and understanding what portions would need to be rewritten if porting the kernel to different platforms.

### **COMPILER**

Understanding the various optimizations that the compiler offers or performs is necessary in order to produce real-time systems which meet the performance criteria. The real-time system when written in C relies heavily on the compiler to produce compact and efficient code. The compiler that NASA utilizes is purchased from MicroWare. The reason for using this compiler is that it will generate object code in the memory module format. MicroWare is now offering a compiler which provides a great deal of optimization. Even though NASA has not yet purchased this compiler, the compilations offered by this new compiler will be studied because NASA has plans to purchase the new compiler. The optimizations provided by the compiler are key to generating a fast and compact kernel from the C language version. Traditionally, compilers would simply translate C code into assembly code, but now compilation is done in stages where front end module compiles source code into an intermediate code (I-code). This intermediate code is independent of

the source language and target microprocessor. Since compilation is done in phases, each phase is capable of performing optimizations. Such phase-oriented optimizations include front end, intermediate code, back end, and assembler optimizations.

The front end phase of compilation is responsible for compiling source code into intermediate code. The optimizations associated with this stage are translation-type optimizations which include loop rotation and constant folding. Loop rotation refers to reducing the complexity of the branching operations associated with do, while, and for loops. Constant folding entails reducing operators whose operands are constants to the result of the operator.

The intermediate code (I-code) phase uses the I-code from the front end stage and links it with other libraries to produce an I-code image of the entire application. The I-code optimizer will then process the I-code image of the entire application. The optimizations that are provided at this stage are: assignment translation, code motion and combining, common subexpression elimination, constant folding, constant propagation, loop invariant hoisting, loop unrolling, pointer tracking, procedure in-lining, useless code elimination, variable lifetime, and variable strength reduction. Assignment translation refers to changing normal assignments into more efficient assignment operators wherever possible. Code motion and combining means that multiple copies of the same code is moved to a common location. Common subexpression elimination results in the elimination of the recomputing of identical expressions. The constant folding optimization performed by the I-code linker determines the span of a local variable in a function and will free the storage when the variable is not being used. The optimization related to constant propagation results in having variable references which are known to contain a constant value changed to constants. Loop invariant hoisting will relocate invariant code from inside to the

outside of a loop. Loop unrolling attempts to reduce the overhead of having to increment and check the loop induction variable by making more copies of a loop body. Pointer tracking examines the contents of pointer variables when they are dereferenced in order to best determine their effect on common subexpressions. Procedure in-lining is used to replace a call to a function with a copy of the function. Useless code elimination will delete any useless assignments and will compute the right hand side until all useless assignments have been eliminated. Variable lifetime provides additional information to the I-code in order to allow the back end optimization to make the most efficient use of the target register set. The variable strength reduction optimization will replace multiplications involving a loop induction variable with a series of additions.

The back end stage takes the I-code and translates it into assembly language. The optimization techniques that can be applied at this stage are the dynamic register set, register coalescing, register coloring, and data area layout. The dynamic register set technique will allocate registers for local variables that are different for each function. The register coalescing performs operations such that the result ends up where it will be needed. Register coloring determines how the registers will be best used within a function; the variables that are least used will be placed on the stack. The data area layout is responsible for placing the most frequently referenced global data in the fastest storage area.

The assembly obtained from the back end stage is further optimized by the assembly optimizer. The optimizations performed by the assembly optimizer are the branch shortening and PC-relative addressing mode shortening. The branch shortening option will reduce the instruction size on the branch instruction if the distance to the destination

is within certain limits. The PC-relative addressing mode shortening option will reduce the instruction size for the PC-relative addressing mode if the label is within certain limits.

Although many optimization options are offered at the various stages, the user can select which options are best suited for the application.

Unlike most compilers, which provide optimization on a file-by-file basis, this compiler applies optimization techniques over the entire application. This is accomplished by linking at the intermediate code level and at the assembler level.

## **REWRITING PCOS**

Since the kernel is the major element of PCOS, converting the kernel to C before converting the other elements of PCOS was considered to be the most valuable option. Another factor, which led to converting the kernel to C before converting the other modules, was that the kernel was not as hardware dependent as the other modules in PCOS. The modifications to the kernel have already been discussed, but the modifications concerning other modules that the kernel interfaces need to be analyzed. Also included in the analysis is what other modules should also be converted to C.

The conversion of the kernel required either slight modifications to the other modules or to the interface to the other modules. For instance, the kernel's init routine originally received information in certain designated registers. These registers were loaded by the

bootstrap module. The bootstrap module was changed such that it passes to the kernel a pointer to a structure. This structure contains the information needed by the kernel's init routine. Another modification that resulted from converting the kernel dealt with the interface between the kernel and the debugger services. A jump table was instituted in the kernel so that a pointer to the debugger service function is utilized when accessing any debugger service.

Although the kernel in comparison with the other system level modules is probably the least hardware dependent, portions of the other modules which include the bootstrap, the debugger services, the file managers and the device drivers could also be rewritten in C. The hardware dependent sections of these modules could be left in assembly.

Another problem encountered while developing a new version of the kernel was that portions of code appeared to be superfluous. For example, there were additional entries in the table which list the system services for which there was no actual system service function.

## **PROBLEMS WITHIN THE KERNEL**

The new C version of the kernel contains certain corrections to address known problems that have existed in one, two, or all three assembly versions of the kernel. These previously reported problems have been found in the various segments of the kernel; thereby, they are not restricted to any one functional segment of the kernel.

The errors that existed in the maintenance of memory modules pertain to the way in which the kernel unloaded and unlinked modules. The assembly versions erroneously did not provide a proper counterpart to the load function. The unload module would perform the same as the unlink function. This means that link count associated with a memory module would simply be decremented to zero but would not be removed from memory once its link count reached zero. The correction entailed providing an unload function which would remove the module from memory if the corresponding link count was already zero. The unload function will not modify the link count. The unlink function would simply decrement the link count associated with a module.

The implementation of timers was somewhat flawed. The old configuration consisted of groups of timers with each timer having a count of ticks. The configuration was changed to reduce the time needed to decrement the counters. Instead of having to decrement many counters, only one timer counter is updated. This counter is at the head of the timer group table. The rest of the table entries consist of delta times. This delta time is defined as the difference between the desired timer period and the timer count. The timers are placed in a linked list in increasing order of delta times.

Another segment of the kernel which needed correction was the scheduling portion. Certain inefficiencies existed in the tick routine that were corrected. For instance, if there is only one active process in the queue, the complete scheduling algorithm is executed. Such an execution results in saving the state of the current process, sorting the active queue, and placing the process at the head of the active queue. If there is only one process currently active, a great deal of overhead is created. The tick routine was

modified to account for situation where only process is active. The active process queue is now checked prior to executing the complete scheduling algorithm.

A problem which was not specific to any one portion of the kernel, but instead existed in all areas was the problem of the notifying the application of an error. Even though the kernel may have encountered errors during its processing, the application may have never been notified. The reason why processes were not properly notified of errors was that register containing the number associated with the corresponding error condition was being cleared accidentally as the kernel was passing control back to the process. The correction that was implemented utilizes an environment variable, `errno`, which contains the corresponding error number.

## **PORTING ISSUES**

Although the new C version of the kernel has been designed to be portable, there are certain portions that are still platform-specific. Even with operating systems that are highly portable such as UNIX, the hardware dependent portions are identified and must be rewritten when they are ported to the various platforms. In fact about 10% of UNIX needs to be rewritten when porting to a different platform. The reason for this is that portions of the kernel are dependent on the hardware. The kernel that was developed is to be used in conjunction with the Motorola 68000 family of processors. The portions of the kernel that need to be rewritten are the `syscall`, `semaphore`, and `exceptions` routines.



Identifying the specific segments of the kernel that are platform-specific and the reasons for the dependence will make it easier to port to various platforms.

The syscall function which processes all system service requests relies on the trap 0. Such reliance on a trap makes the function hardware dependent. When porting to different platforms, it becomes necessary to implement this function using some form of software interrupt or trap. Also this function is responsible for saving the state of process meaning the contents of the registers must be saved. Converting to a different platform also involves analyzing to which extent the register sets are to be saved in order to correctly capture the state of the process.

The semaphore routine which is part of interprocess communications mechanisms will also require some modification when the kernel is ported to a different platform. The reason for this mechanism being platform-specific is that its implementation relies on a non-divisible instruction such as test and set (tas) instruction. The tas instruction utilizes the read-write-modify cycle offered by the Motorola family of processors. When porting the semaphore routine, an equivalent non-divisible instruction must be used.

The exceptions routine contains functions that are processor-specific. In general, the handling of exceptions is dependent on the processor. This is consistent with portions of the exceptions routine being dependent upon the platform. If the kernel is to be ported to a different platform, the functions that are found in the exceptions routine would either have to be rewritten to accommodate the new processor or removed if there is not an equivalent exception in the new platform.

Although the specific segments of the kernel that are hardware specific have been identified, there are segments of the kernel which rely on the functions to disable and enable the interrupts. These functions are simply assembly routines which mask interrupts in the status register of the processor. These functions will also have to be rewritten in order to inhibit and enable interrupts when moving to a different platform.

The other portions of PCOS that also need to be rewritten when moving from one platform to another are the bootstrap, debugger services, file managers, and device drivers. These portions are even more hardware specific.

## **CHAPTER 6**

### **RECOMMENDATIONS**

Once the kernel was developed, several recommendations can be made as to how to further enhance, improve, and test the kernel. These recommendations include using benchmarking techniques to determine the performance specifications of the kernel, analyzing the POSIX standards and verifying that the kernel is capable of meeting such standards. Another facet that merits attention would be to determine how practical would a C++ version of the kernel be.

### **FUTURE ENHANCEMENTS**

Although the conversion of the PCOS kernel from assembly to C included correcting certain existing problems, enhancements to the PCOS kernel could also be made that would be most beneficial. These enhancements would probably be most useful in the area of interprocess communications, but the memory management arena could also stand some improvement.

The facilities which are concerned with interprocess communications could certainly be enhanced. Such enhancements may include the dynamic creation of mechanisms, the upgrading of the semaphore, the queueing of signals, and the use of events in conjunction with the queue system.

An improvement concerning these facilities would be the dynamic creation of such mechanisms instead of relying on the init routine to create these mechanisms based on the desired amount designated in the init module. For instance, the number of mailboxes, semaphores or queues that are needed is listed in the init module. The init routine would create the number of mailboxes and queues according to the init module. This current method is somewhat restrictive. The ability to dynamically create such mechanisms would certainly increase the flexibility of PCOS. The mechanism that would most benefit from such an improvement would be the queue system which is used extensively throughout the current applications.

Upgrading the semaphore system would also include the use of the counting semaphore instead of a binary semaphore. A binary semaphore accepts only two values, a zero or a one. A counting semaphore allows a multitude of values ranging from zero to the size of the semaphore.

The queueing of signals would be beneficial to PCOS. With the current implementation, signals may be lost. For instance, if a signal is sent to a process which is in the middle of processing another signal, the second signal will not be processed or seen by the designated process. To overcome such problems, the signals could be queued such that signals could not be ignored or lost.

Another enhancement that would be of great use to PCOS will be the use of events to signify when a queue is ready for processing. With the current configuration, a process desiring to enqueue a message to an already full queue must either be placed in a waiting to enqueue list or continually poll the queue until there is enough room in the desired queue. The placing of queue in a waiting list or polling the queue also applies to the

instances where a process wants to dequeue a message from an empty queue. In this case, the process must either wait or poll the queue until a message is placed in the desired queue. If the usage of events is incorporated into the queue system, the waiting on a queue or polling a queue could be eliminated. An event could be sent to a process to indicate that the once empty queue now has data to be dequeued or the once full queue now has room for more messages.

An enhancement to the memory management system may include the implementation of colored memory. A colored memory system would allow the kernel to recognize different types of memory and to reserve portions of memory for specific purposes. The kernel could regulate and isolate access of such memory.

## **BENCHMARKING**

Even though there are several benchmarking programs available to analyze the performance of software, there is very little available in terms of benchmarking in conjunction with real-time systems. The general benchmarking facilities that are available include the SPEC benchmark, Whetstone, and Dhrystone. The SPEC benchmark is used for comparing the performance of workstations. The SPEC benchmark uses pieces of code from functional applications which are representative of an application class in order to check how the workstation makes use of the system resources. The Whetstone and Dhrystone consist of test programs which are created in order to represent the usage of the system resources. These two benchmarks are sometimes referred to as synthetic

benchmarks. These generic benchmarks do not test the characteristics associated with real-systems, namely, context switching and interrupt latency.

In terms of testing real-time systems, there are only two proposed benchmarks: the Rhealstone benchmark and the Hartstone benchmark. The Rhealstone benchmark attempts to address the problems associated with measuring real-time performance by providing six categories of functions which are commonly provided by real-time systems. These six components are preemption, task-switching, semaphore shuffling, deadlock breaking, intertask message latency, and interrupt latency.

The metric designed to measure preemption is responsible for measuring the time the kernel takes to switch between processes which do not have equal priorities. This metric will measure the time required by a higher priority to preempt a lower priority task. The component of the benchmark which measures the time it takes for the kernel to switch between two active tasks of equal priority is the task-switching test. Both the preemption and the task-switching tests address the issue of the context-switching times. The differences between these two tests is the priority resolution. Priority resolution is considered an integral part of the scheduling mechanism.

The two metrics which pertain to resource allocation are the semaphore shuffling and the deadlock breaking metrics. Semaphore shuffling identifies the overhead associated with passing the ownership of the semaphore from one process to another. The deadlock breaking metric measures the time to break to a specific deadlock. The deadlock breaking metric seems rather useless because very few kernels including PCOS offer mechanisms for breaking deadlocks. In fact, it would more practical to develop a metric which would measure other resource allocation functions.

The Rhealstone metric which analyzes how long it takes for a process to pass a message to another process is called the intertask message latency. This metric was originally called the datagram throughput metric. It will analyze how efficient the message passing mechanism is. The passing of messages is one of the facilities that is used for interprocess communications. There is not a specific metric for measuring the time associated with the other interprocess communications facilities such as the event system.

The Hartstone Benchmark consists of a series of five tests which are designed to measure the ability of the real-time system to meet its deadlines. The series of five tests include the periodic harmonic series, periodic nonharmonic, periodic harmonic with synchronization, periodic harmonic with aperiodic processing, periodic harmonic with synchronization and aperiodic processing. This series of tests is responsible for increasing the system loading until the system cannot meet a deadline. Increasing the system loading can be accomplished by increasing the frequency of the highest frequency process, increasing the frequency of all processes, increasing the workload of all processes, and increasing the number of processes.

## **POSIX**

In order to simplify and ease the porting of applications from one platform to another, the IEEE Portable Operating System Interface (POSIX) group is attempting to the develop a standard (IEEE 1003); this standard is to cover a wide range of operating systems services

which include the basic functionality, networking, security, and real-time facilities. The proposals revolve around the standardization of the interfaces to UNIX. The reason why UNIX was chosen is due to its widespread use across a variety of platforms. The problem with using UNIX is that UNIX was never conceived as a real-time operating system. UNIX does not address real-time concerns such as determinism. The POSIX group is attempting to address this concern by having a separate workgroup address the unique requirements associated with the real-time issues (1003.4). The 1003.4 group will attempt to standardize the facilities needed for real-time support. In order to accomplish this, the work was divided into three categories: the processing model, the handling of I/O, and repairs needed for UNIX to support real-time requirements.

Although POSIX is based on UNIX, the basic processing model is somewhat different. These differences include scheduling, timers, and threads. The standard UNIX does not offer a defined scheduler; it is simply a time-sharing system. The scheduling of real-time processes requires that a defined scheduling mechanism be provided. The 1003.4 proposal defines the scheduling interface. The proposal only defines the interface and not the scheduling algorithms. The currently supported scheduling mechanisms are first-come-first-serve and round robin.

The use of timers is offered by POSIX. The Berkeley version of UNIX offers a similar mechanism known as interval timers. Such a mechanism allows a process to set up a timer to expire either once or periodically; thereby, facilitating time-driven processing.

The implementation of threads which can be considered to be multiple flows of control within a single address space differs greatly from the UNIX processing model where a process is considered a single flow of control within a protected address space. With



UNIX, the sharing of data between process is difficult, yet real-time applications require a great deal of coordination and cooperation between threads of control. POSIX needs a mechanism to coordinate and communicate between threads. Such a mechanism requires indivisible memory instructions such as a test-and-set instruction. A counting semaphore has been proposed as the mechanism for implementing coordination and cooperation between threads. Originally, the 1003.4 group had proposed a binary semaphore, but they have now proposed a counting semaphore which is the more general application of the semaphore.

The handling of I/O is another area that is being investigated from a real-time perspective. The standard UNIX implements synchronous I/O. This means that the calling process is blocked until the data have been queued to be written to the desired device. UNIX offers no method of forcing I/O to circumvent the buffer cache. POSIX offers the real-time files mechanism where a thread can bypass the buffer cache and mandate that all I/O is to be completed synchronously. This means that all I/O is transferred to the designated device before the I/O operation is complete. POSIX not only offers synchronous I/O but also provides an asynchronous I/O facility. Asynchronous I/O allows a calling thread to continue its work. POSIX notifies the calling thread of the completion using a signal.

POSIX includes a number of facilities that are targeted toward repairing the UNIX system in order to support the real-time requirements. These facilities include memory locking, enhancing the signal mechanism, and message passing.

The memory locking facility is used for locking the context of a process in physical memory and not allowing the context to be swapped out. UNIX usually behaves as a swapping paging system. This means that an process may have its context swapped out to

disk when another process needs the memory. This swapping is not acceptable in real-time systems so POSIX offers the ability to lock memory that an application does not get swapped.

Although UNIX offers a signal facility whereby a process can be asynchronously notified of an occurrence, this facility has not been very reliable. The problem with the signal mechanism arises when multiple signals are issued to the same process. Since UNIX offers no queueing of these signals, signals may be lost. The POSIX proposal requires that signals be queued.

The passing of messages is another area which POSIX is analyzing. The mechanisms that UNIX offers for message passing are pipes and sockets. Pipes pass raw streams of bytes between processes; these messages are unstructured. Sockets are part of the networking and pass more structured messages which adhere to variety of network protocols. POSIX proposal a message passing facility that provides a mechanism whereby structured messages can be sent between processes.

Certain operating systems may have real-time POSIX features, but this does not necessarily guarantee real-time performance. POSIX functionality is required while performance is beyond the scope of the standard. Although performance is not considered to be part of the standardization effort, the POSIX 1003.4 group did recognize the importance of performance metrics. The 1003.4 group has been the only working group even remotely concerned with performance metrics. This group has proposed a set of performance measurements that a system may provide. Providing this set of measurements with the system has been left as an option for the real-time vendors.

## C++

The current trend in terms of software development is to migrate toward an object-oriented environment. This means that software may be developed using C++ instead of C. This trend has been seen in the development of application software and not in the development of operating systems. The application software that is being developed in C++ has not been in the real-time domain but rather in graphics or general purpose applications where time is not issue. Although C++ offers many advantages that C does not, there is a great deal of overhead associated with C++. Such overhead may be the reason why the development of operating systems has not migrated toward using C++. Not even UNIX which is more forgiving of problems with overhead has migrated to C++. Overcoming such problems must take place before development of conventional operating systems is done using C++. The method which will probably result in overcoming such impediments is compiler optimization. Once conventional operating systems are developed using C++, the development of real-time operating systems using C++ will probably follow. Although such development is still in future, it is important to understand what the features of C++ offer. The development of real-time applications in C++ will probably precede the development of real-time operating systems in C++.

One of the greatest problems that impedes the development of C++ real-time applications is the issue of run-time penalties which are associated with the features of C++. These penalties to a certain extent may even be reduced by using certain compiler optimizations.

There are certain rather minor features of C++ that do not have any performance penalties. Such features include function prototypes, the type-checking rules, factors that

affect the visibility or accessibility of names, or the mixing of declarations and statements. The function prototypes and type-checking rules can also be found in C especially ANSI C. C++ does offer stricter type-checking rules. An example of a factor that affects the visibility or accessibility of names can be consider the scope resolution operator. The mixing of declarations and statements is a feature of C++ which provides the flexibility of having declarations in the middle of the function. The declaration must be before the variable is used, but it can be after other statements. This means that variable declarations are not restricted to being at the top of a function.

The other minor features of C++ which may have either a minor impact on performance or may require additional space involve the use of global variables and templates. Initialization of global variables at run-time results in delaying the program start-up. Such initialization pertains to non-constant initialization expressions for global variables. The use of templates requires additional space each time a template function is called using different argument types.

The features that make C++ so powerful as a language, especially in the area of object oriented design, are the use of classes, function or operator overloading, and inheritance. The use of classes is an integral part of C++. The closest approximation of a class in C is considered the structure. The class includes both members and member functions unlike the C structure which consists of only members. The overhead due to the classes is related to use of constructor and destructor functions. The constructor function provides the complete control over the allocation of abstract data types while the destructor function provides the deallocation of these abstract data types. The use of virtual member functions also adds overhead because the compiler cannot determine which function will

actually be used. The compiler simply builds the virtual function tables; these tables are maintained during run-time.

Function or operator overloading is extremely useful in C++. Function overloading allows several functions to have the same name but different argument lists. This feature has no run time penalty because the linker and compiler are responsible for matching the function calls with the corresponding definition. Operator overloading may be considered a type of function overloading. Operator overloading refers to redefining the operator. Operator overloading essentially does not provide any run time penalty, but certain caution must be exercised. The efficiency of the operator function is dependent upon how efficiently a compiler implements structure-returning functions. Some compilers may not effectively implement structure-returning functions.

Inheritance is used in defining derived class types. Inheritance, itself, does not incur a run time penalty, but much care is needed in its implementation. C++ compilers have much freedom in the implementation of inheritance. This means that implementing the derivation of classes may result in overhead.

## REFERENCES

- Anderson, K., "The best C optimizer is still between your ears", *Personal Engineering & Instrumentation News*, April 1991, pp. 63-67.
- Atkinson, L., & Atkinson, M., *Using C/C++*, Que Corporation, Carmel, IN, 1993.
- Bauer, M. M., "Real-Time Atlas", *Embedded Systems Programming*, April 1991, pp. 56-62.
- Barrett, T., & Wood, M., "Inside a Real-Time Kernel", *Embedded Systems Programming*, November 1990, pp. 34-47.
- Bromley, J., "Real-Time Conflicts", *Embedded Systems Programming*, November 1990, pp. 22-31.
- Brown, R., "Mix C and assembly language for fast real-time control", *EDN*, Software Engineering Special Issue 1990, pp. 41-47.
- Clements, A., *Microprocessor Systems Design*, PWS-KENT Publishing Company, Boston, MA, 1987.
- Deitel, H.M., *Operating Systems*, Addison-Welsey Inc., Reading MA, 1990, 2nd ed.
- Duntemann, J., "Chimney-Pipe", *Dr. Dobb's Journal*, August 1991, pp. 157-163.
- Ganssle, J. G., "Perils of the NMI", *Embedded Systems Programming*, April 1991, pp. 79-81.
- Gallmeister, B. O., "Real-Time POSIX", *Embedded Systems Programming*, October 1992, pp. 28-40.
- Halbert, J., "Embedded Real-Time Multitasking Kernel", *The C Users Journal*, March 1991, pp. 33-47.
- Harbison, S. P., "C++ and Real-Time Performance", *Embedded Systems Programming*, July 1993, pp. 18-26.
- Husain, K., "Real-Time Buffering", *Embedded Systems Programming*, April 1991, pp. 36-42.
- Kernighan, B., & Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

- Koball, B., "A Real-Time Lexicon", *Embedded Systems Programming*, February 1990, pp. 23-28.
- Koball, B., & Novickis, A., "Shootout at the RT Corral", *Embedded Systems Programming*, September 1990, pp. 44-55.
- Kochan, S., & Wood P., *Topics in C Programming*, John Wiley & Sons, Inc., New York, 1991, Revised ed.
- Labrosse, J., "A Real-Time Kernel in C", *Embedded Systems Programming*, May 1992, pp. 40-53.
- Labrosse, J., "Implementing a Real-Time Kernel", *Embedded Systems Programming*, June 1992, pp. 44-49.
- Labrosse, J., "Understanding Semaphores", *Embedded Systems Programming*, October 1990, pp. 42-50.
- Lethaby, N., & Black, K., "Memory Management Strategies for C++", *Embedded Systems Programming*, July 1993, pp. 28-34.
- Matthews, P., & Furht, B., "Evaluating Real-time UNIX", *Embedded Systems Programming*, July 1991, pp. 28-36.
- Merriam, R., "Performance vs. Paradigm", *Embedded Systems Programming*, March 1991, pp. 18-31.
- Merriam, R., "68000 C Cross Compilers", *Embedded Systems Programming*, November 1990, pp. 61-69.
- Micrology pbt, Inc, edited & compiled, *VMEbus Specification Manual*, PRINTEX Publishing, Inc., 1985.
- Moore, R., "Preemptive Virtues", *Embedded Systems Programming*, February 1991, pp. 24-30.
- Motorola Inc., *M68000 8-/16-/32-Bit Microprocessor User's Manual*, Prentice Hall, Englewood Cliffs, NJ, 1990, 8th ed.
- Motorola Inc., *M68000 Family Reference*, 1989, 2nd printing.
- Motorola Inc., *M68000 Programmer's Reference Manual*, 1992.

- Motorola Inc., *M68020 Microprocessors User's Manual*, 1992.
- Motorola Inc., *M68030 Microprocessor User's Manual*, 1992.
- Motorola Inc. *M68040 Microprocessor User's Manual*, 1992.
- Plauger P. J., "Designing Device Handlers", *Embedded Systems Programming*, April 1991, pp. 87-92.
- Plauger P. J., "Evaluating Real-Time Operating Systems", *Embedded Systems Programming*, February 1990, pp. 30-36.
- Plum, T., & Brodie, J., *Efficient C*, Plum Hall Inc., Cardiff, NJ, 1985.
- Rathje, E. J., "Smooth Sailing on a RISCy C", *ESD*, May 1989, Reprint.
- Ripps, D.L., *An Implementation Guide to Real-Time Programming*, Yourdon Press, Englewood Cliffs, NJ, 1989.
- Ripps, D. L., "The nature of real time", *EDN*, Software Engineering Special Issue 1990, pp. 29-34.
- Ripps, D. L., "The multitasking mindset meets the operating system", *EDN*, October 1, 1990, pp. 115-123.
- Ripps, D. L., "Developing real-time requirements", *EDN*, October 11, 1990, pp. 223-228.
- Ripps, D. L., "Understanding the complexity of tasking in real time", *EDN*, October 25, 1990, pp. 193-199.
- Ripps, D. L., "Basic task services for real-time execution", *EDN*, November 8, 1990, pp. 249-258.
- Ripps, D. L., "Time and time of day", *EDN*, November 22, 1990, pp. 197-202.
- Ripps, D. L., "Events flags", *EDN*, January 3, 1991, pp. 115-120.
- Ripps, D. L., "Message buffers and mailboxes", *EDN*, January 21, 1991, pp. 115-124.
- Ripps, D. L., "Semaphores and controlled shared variables", *EDN*, February 4, 1991, pp. 85-90.
- Ripps, D. L., "Task coordination and communication via signals", *EDN*, February 18, 1991, pp. 157-162.



- Ripps, D. L., "Task coordination: specific methods, general principles", *EDN*, March 1, 1991, pp. 97-110.
- Schildt, H., *The Art of C: Elegant Programming Solutions*, McGraw-Hill, Berkeley, CA, 1991.
- Sergeant, F., "Design Tradeoffs", *Embedded Systems Programming*, April 1991, pp. 24-34.
- Silberschatz, A., Peterson, J. & Galvin P., *Operating System Concepts*, Addison-Welsey Inc., Reading MA 1991, 3rd ed.
- Singh, I., "Inder Singh on: Posix", *Computer Design*, March 1, 1991, pp. 23-25.
- Skinner, E., "VMEbus for Software Engineers", *Embedded Systems Programming*, June 1993, pp. 24-44.
- Small, C. H., "Real-time UNIX & UNIX look-alikes", *EDN*, June 7, 1990, pp.88-102.
- Sperry, T., "Rheality Checks", *Embedded Systems Programming*, February 1990, pp. 9.
- Stankovic, J.K. & Ramamritham, K., *Hard Real-Time Systems*, Computer Society Press of the IEEE, Washington, D.C., 1988.
- Weiss, R., "RISC chips muscle into embedded applications", *EDN*, August 5, 1993, pp. 45-52.
- Williams, T., "Real-time multiprocessing pushes software limits", *Computer Design*, November 1991, pp. 63-70.
- Williams, T., "Real-time Unix develops multiprocessing muscle", *Computer Design*, March 1, 1991, pp. 26-30.
- Wood, M. & Barrett, T., "Real-Time Primer", *Embedded Systems Programming*, February 1990, pp. 20-28.
- Wynia, T., "RISC and CISC Processors Target Embedded Systems", *Electronic Design*, June 27, 1991, pp. 55-70.

## **APPENDIX**

### **SOURCE CODE**

```

#include <errno.h>      /* errno defines */
#include <dir_page.h>   /* direct page header */

#define AUXMASK          0x00644800
#define MAINMASK        0x0d000000

/* send a signal to a process */

send( rec_pid, sig_code)
u_int32 rec_pid;      /* receiver's process id number */
u_int32 sig_code;     /* signal code */
{
proc_desc *pd;

disable irq;

/* obtain process descriptor based on the base address of the
   process descriptors and process id number */
pid--;                /* using pid as offset */
pd = d_pdba + (pid*P_SIZE)

/* verify process descriptor is valid */
/* descriptor is invalid if id is negative, if descriptor is not
   within range, or if process is dead */
if ( (pid < 0) || (pd > d_pdbae) || (pd->state & DEAD) ) {
    errno = E_IPRID; /* errno indicates invalid process id */
    return (FAIL);
}

/* determine if process is active or inactive */
if ( pd->state & AUXMASK ) {
    pd->auxpque->auxnque = pd->auxnque;
    if ( pd->auxnque == NULL )
        pd->auxpque->auxnque = pd->auxpque;
    activateproc();
}
if ( pd->state & MAINMASK ) {
    pd->pqueue->nqueue = pd->nqueue;
    if ( pd->nqueue == NULL )
        pd->pqueue->nqueue = pd->pqueue;
    activateproc();
}

/* check if signal already present */
/* if not present, save else indicate error */
if ( sig_code != WAKE ) {
    if ( pd->signal == NOTPRESENT || sig_code == KILL ) {
        pd->signal = sig_code;
    }
    else {
        errno = E_USIGP;      /* signal already present */
        return (FAIL);
    }
}

```

```

        }
    }
}

/* wait puts the current process in the process queue */
wait()
{
    disable_irq;

    if ( pd->cid == 0 ) {
        errno = E_NOCHILD;    /* current process has no child */
        return (FAIL);
    }

    pd->state &= STCPRESERVE;    /* clear most states */
    pd->state != WAIT;          /* indicate a WAIT */

    /* if wait queue empty, set next queue ptr to empty */
    if ( d_wprocq == NULL )
        pd->nqueue = NULL;
    else {
        /* if not empty, adjust ptrs */
        d_wprocq->pqueue = pd;
        pd->pnueue = d_wprocq;
    }

    /* insert process in wait queue */
    d_wprocq = pd;

    /* pd->pqueue = &(d_wprocq) - &(0->nqueue); setup fake ptr */
    pd->pqueue = FAKEQHD (d_wprocq, nqueue);
    nextproc();    /* continue processing */

}

/* sleep puts current process in the sleep queue for a
   specified amount of ticks. if ticks is zero, this means
   sleep indefinitely. */

sleep (ticks)
u_int32 ticks;    /* amount of time to sleep. if 0, indefinitely */

{
    proc_desc *ptr;

    disable_irq;

    pd->state &= STCPRESERVE;    /* clear previous process state */

```

```

if ( ticks == 0 ) {
    pd->state |= INDEFSLEEP;          /* indicate indefinite sleep */
    pd->nqueue = d_isprocq; /* set process as head */
    pd->pqueue = FAKEQHD (d_isprocq, nqueue);
    d_isprocq = pd;

    /* are there other processes in indefinite sleep? */
    if ( pd->nqueue != NULL )
        pd->nqueue->pqueue = pd;

    nextproc();          /* let next process run */
}
pd->state != TIMESLEEP;          /* indicate a timed sleep */

/* if sleep queue is empty, set up as head */
if ( d_sprocq == EMPTY ) {
    pd->nqueue = NULL;
    pd->pqueue = FAKEQHD (d_sprocq, nqueue);
    d_sprocq = pd;
    nextproc();
}

/* sleep queue is arranged in increasing tick order so
   traverse queue looking for where to insert process */
for ( ptr = d_sprocq; ptr->nqueue != NULL; ptr = ptr->nqueue ) {

    if ( ptr->stickcnt > ticks ) {

        /* if ticks is less than first queued process tick
           count, establish a new queue head */
        if ( ptr = d_sprocq ) {
            pd->pqueue = FAKEQHD (d_sprocq, nqueue);
            pd->nqueue = ptr;
        }
        else { /* insert in middle of sleep queue */
            ptr->pqueue->nqueue = pd;
            pd->nqueue = ptr;
            pd->pqueue = ptr->pqueue;
            ptr->pqueue = pd;
        }
    }
    else {
        if ( ptr->stickcnt = ticks ) {
            pd->nqueue = ptr->nqueue;
            pd->pqueue = ptr;
            ptr->nqueue = pd;
            ptr->nqueue->pqueue = pd;
        }
    }

    nextproc;
    return;
}

```

```

    }
    /* insert at the end of the sleep queue */
    ptr->nqueue = pd;
    pd->pqueue = ptr;
    pd->nqueue = NULL;
}

/* post will place mail in a mailbox */
post (box, mail_val, flag)

u_int32 box;          /* mailbox number */
u_int32 mail_val;    /* mail value to post*/
u_int32 flag;        /* flag indicating a wait or no wait
                    if there is mail in the mailbox */

{
    struct mbox *mail_box;
    proc_desc *ptr;

    disable_irq;

    /* if there is no mail, indicate a failure */
    if (mail_val == NOMAIL) {
        errno = E_NOMAIL;    /* there is no mail to send */
        return (FAIL);
    }

    mail_box = d_postoff + box * (sizeof(mbox));

    /* verify given box number is a valid mailbox */
    if ( mail_box < d_postoff ) !! ( mail_box > d_postoffe ) {
        errno = E_BADMB;    /* invalid mailbox number */
        return (FAIL);
    }

    /* is there mail in the mailbox */
    while ( mail_box->mail != EMPTY ) {

        /* check if need to wait if there is mail */
        if ( flag == WAIT ) {

            /* check if someone is waiting for mail */
            if ( mail_box->pqueue == NULL ) {
                /* setup as head of queue since no one is waiting */
                pd->auxpque = FAKEQHD(mail_box->pqueue, auxnque);
            }
            else { /* processes waiting to post mail to this mail_box */
                /* find the end of the queue */
                for (ptr=mail_box->pqueue; ptr->auxnque != NULL; )

```

```

        ptr = ptr->auxnque;
        /* put at the end of queue */
        pd->auxpque = ptr;
    }
    mailbox->pqueue = pd;
    pd->auxnque = NULL;

    /* show that process is not active and is waiting to post mail */
    pd->state &= ^ACTIVE;
    pd->state |= POSTWAIT;
    nextproc();

else { /* flag is a no wait */

    errno = E_MBBUSY; /* mailbox is busy */
    return (FAIL);
}
/* was a kill signal received */
if ( pd->signal == KILL ) {

    /* post mail if there is mail and
    some one wants to post mail before
    termination */
    if ( mail_box->mail == EMPTY
        && mail_box->pqueue != NULL ) {
        ptr = mail_box->pqueue;
        mail_box->pqueue = ptr->auxnque;
        if ( ptr->auxnque != NULL ) {
            ptr->auxnque->auxpque = FAKEQHD (ptr->auxnque, pqueue);
            ptr->auxnque = NULL;
            ptr->auxpque = NULL;
            activateproc();
        }
        exit( pd->signal );
    }
}

mail_box->mail = mail_val;
if (mail_box->gqueue != NULL) {
    ptr = mail_box->gqueue;
    mail_box->gqueue = ptr->auxnque;
    if ( ptr->auxnque != NULL ) {
        ptr->auxnque->auxpque = FAKEQHD (ptr->auxnque, gqueue);
        ptr->auxnque = NULL;
        ptr->auxpque = NULL;
        activateproc();
    }
}

/* getmail will retrieve mail from a mailbox */

```

```

getmail (box, &mail_val, flag)

u_int32 box;          /* mailbox number */
u_int32 *mail_val;   /* mail value that was retrieved */
u_int32 flag;        /* flag indicating a wait or no wait
                      for mail to arrive */

{
struct mbox *mail_box;
proc_desc *ptr;

disable_irq;

mail_box = d_postoff + box * (sizeof(mbox));

/* verify given box number is a valid mailbox */
if ( mail_box < d->postoff ) {
    errno = E_BADMB; /* invalid mailbox number */
    return (FAIL);
}

/* is there mail in the mailbox */
while ( mail_box->mail == EMPTY ) {

    /* check if need to wait for mail to arrive */
    if ( flag == WAIT ) {

        /* check if someone is waiting for mail */
        if ( mail_box->gqueue == NULL ) {
            /* setup as head of queue since no one is waiting */
            pd->auxpqe = FAKEQHD(mail_box->gqueue, auxnque);
        }
        else { /* processes waiting to get mail from this mail_box */
            /* find the end of the queue */
            for (ptr=mailbox->gqueue; ptr->auxnque != NULL; )
                ptr = ptr->auxnque;
            /* put at the end of queue */
            pd->auxpqe = ptr;
        }
        mailbox->gqueue = pd;
        pd->auxnque = NULL;

        /* show that process is not active and is waiting to post mail */
        pd->state &= ^ACTIVE;
        pd->state |= GETMAILWAIT;
        nextproc();
    }

    else { /* flag is a no wait */

        errno = E_MBBUSY; /* mail box is busy */
        return (FAIL);
    }
}

```



```

/* was a kill signal received */
if ( pd->signal == KILL ) {

    /* get mail if there is any in the mail_box and
       some one wants to retrieve the mail before
       termination */
    if ( mail_box->mail != EMPTY
        && mail_box->gqueue != NULL ) {
        ptr = mail_box->gqueue;
        mail_box->gqueue = ptr->auxnque;
        if ( ptr->auxnque != NULL ) {
            ptr->auxnque->auxpqe = FAKEQHD (ptr->auxnque, gqueue);
            ptr->auxnque = NULL;
            ptr->auxpqe = NULL;
            activateproc();
        }
        exit( pd->signal );
    }
}

*mail_val = mail_box->mail;
mail_box->mail = EMPTY;
if (mail_box->pqueue != NULL) {
    ptr = mail_box->pqueue;
    mail_box->pqueue = ptr->auxnque;
    if ( ptr->auxnque != NULL ) {
        ptr->auxnque->auxpqe = FAKEQHD (ptr->auxnque, pqueue);
        ptr->auxnque = NULL;
        ptr->auxpqe = NULL;
        activateproc();
    }
}
}

```

```

#include <proc.h>          /* process descriptor header */
#include <errno.h>        /* error number header */
#include <event.h>        /* event definitions header */

/* System calls related to events */

/* initialize a process's event queue */
event_init(p)
proc_desc *p;    /* address of process descriptor */

{

p->eventstr=p->events;    /* beginning of event table */
p->eventin=p->events;    /* save in ptr */
p->eventout=p->events;    /* save out ptr */
p->eventend=numevents*eventsize*4    /* end of event queue */
p->eventcnt = 0;    /* initialize count of events */
p->eventflag = 0;    /* initialize flags */

}

sendevent(incarn, opcode, address, pid)
uint32  incarn;    /* destination process incarnation number */
uint32  opcode;    /* opcode (meaning to users only) */
uint32  address; /* address (meaning to users only) */
uint16  pid;    /* destination process id */

{

disable_irq;    /* inhibit interrupts */

/* obtain process descriptor based on the base address of the
   process descriptors and process id number */
pd = d->pdba + ((pid--)*P_SIZE)

/* verify process descriptor is valid */
if ( pd < 0 || pd > d->pdbae ) {
    errno = E_IPrId; /* errno indicates invalid process id */
    return (FAIL);
}

/* verify that a valid incarnation number was received */
if ( pd->incarn != incarn ) {
    errno = E_Incar; /* errno indicates invalid incarnation # */
    return (FAIL);
}

else
    *(pd->eventin)++ = opcode;
    *(pd->eventin)++ = address;

```

```

/* if at end of event queue wrap using the start */
if ( pd->eventin == pd->eventend )
    pd->eventin = pd->eventstr;

/* check if event queue is full */
if ( pd->eventin == pd->eventout )
    pd->eventflag &= EVENTFULL;

pd->eventcnt++; /* increment event counter */

/* waiting for an event? */
if ( pd->state & EVENTWAIT )
    activateproc(pd);
}

getevent (*address, *count, mode, pd)
long address;          /* event address (meaning to user) */
short count;          /* number of events remaining */
int mode;             /* flag indicating a wait or no wait */
Proc_desc *pd;       /* pointer to process descriptor */

{
long opcode;          /* opcode (meaning to user only) */

disable_irq;         /* inhibit interrupts */

for (;;) {

    if ( !pd->eventcnt ) {
        pd->eventcnt--;
        count = pd->eventcnt;
        opcode = *(pd->eventout)++;
        address = *(pd->eventout)++;

        /* if at the end of event queue, wrap */
        if ( pd->eventout == pd->eventend )
            pd->eventout = pd->eventstr;

        /* indicate that do not have an event */
        pd->eventflag &= (^EVENTFULL);
        return (opcode)
    }

    else {
        if ( mode == WAIT ) {
            errno = E_NoEvents;
            return (FAIL);
        }

        else
            if ( !pd->signal ) {
                errno = E_Signal;
                return (FAIL);
            }
    }
}

```

```
    }  
    else {  
        pd->state != EVNTWAIT;  
        pd->state &= (^ACTIVE);  
        nextproc(pd);  
    }  
}
```

```
/* fork.c comprises the functions necessary for creating a process
   and initializing the process descriptor */
```

```
#include <dir_page.h>
#include <proc.h>
#include <module.h>
```

```
/* fork is responsible for creating a process */
fork(mod_ptr, type_lang, priority, io_paths, &incarn)
module *mod_ptr;
unsigned short type_lang;
unsigned short priority;
unsigned short io_paths;
unsigned long incarn;
{
```

```
prod_desc *proc_d;
int i;
```

```
/* allocate a process descriptor */
if ((proc_d = allprc()) == FAILED)
{
    return(FAIL);
}
```

```
/* was able to allocate a process descriptor */
```

```
/* copy user index from parent */
proc_d->user = pd->user;
```

```
/* copy given priority */
if ((proc_d->prior = priority) == 0)
{
    /* priority not specified, so copy parent's */
    proc_d->prior = pd->prior;
}
```

```
/* clear age */
proc_d->age = 0;
```

```
/* copy default I/O information */
for (i=0; i<DEFIOSIZE; i++)
{
    proc_d->dio[i] = pd->dio[i];
}
```

```
/* check if have a default data ptr */
if (pd->dio != NULL)
{
    /* increment users and static storage count */
    pd->dio->usrs++;
    pd->dio->stat++;
}
```

```

    }

/* check if have a default exec ptr */
if (pd->dexec != NULL)
    {
    /* increment users and static storage count */
    pd->dexec->usrs++;
    pd->dexec->stat++;
    }

/* check if minimum number of paths are specified */
if (iopaths == 0)
    {
    /* since min number not given, set it to three
    for stdin, stdout, & stderr */
    iopaths = 3;
    }

else
    {
    /* verify that the number of paths specified does not
    exceed the max */
    if (iopaths > NUMPATHS)
        {
        errno = E_TMPaths;
        return(FAIL);
        }
    }

/* copy paths */
for (i=0; i<iopaths-1; i++)
    {
    proc_d->path[i] = pd->path[i];
    }

/* set up count of paths */
proc_d->path->cnt = i;

/* initialize process descriptor */
setprc(proc_d, mod_ptr, type_lang);

/* initialize event variables */
eventinit();

/* make current process's youngest child the older sibling to
the new process */

proc_d->os = pd->cid;

/* set parent process id & incarnation number */
proc_d->pid = pd;
proc_d->pincarn = pd->incarn;

```

```

/* get incarnation number for new process */
d_incarn++;
proc_d->incarn = d_incarn;

/* mark new process as the younger sibling of the previously
   youngest child process */
proc_d->os->ys = proc_d;

/* mark new process as the youngest child of the current process */
pd->cid = proc_d;

incarn = d_incarn;

return(proc_d->id);
}

/* setprc initializes the process descriptor associated with the
   newly created process */
setprc(proc_d, mod_ptr, type_lang)
proc_desc *pd;
module *mod_ptr;
unsigned short type_lang;
{

unsigned long mod_entry;
unsigned long mod_hdr;
unsigned long bytes_extracted;
unsigned long *mem_ptr;

/* clear the signal code and signal vector pointer */
proc_d->signal = 0;
proc_d->sigvec = 0;

/* verify that the module is an executable module */
if ((type_lang != M68KOBJ+PRGRM)
    && ( type_lang != M68KOBJ+SYSTEM))
    {
        errno = E_NEMod;
        return(FAIL);
    }

/* link to module */
link(mod_ptr, type_lang, &mod_entry, &mod_hdr);

/* extract some memory */
mem_ptr = extract(mod_ptr->mem+mod_ptr->stack, &bytes_extracted)

/* save pointer to memory and size in the memory block table */
pd->memimg[0] = memptr;
pd->blksiz[0] = bytes_extracted;
}

```

```

/* init finishes off the initialization that began with the
   bootstrap */

#include <init.h>
#include <proc.h>
#include <errno.h>
#include <dir_page.h>

init(i_ptr)
init_struct *i_ptr; /* pointer to information provided by bootstrap */

{

int i;
unsigned long *ptr_numel;
unsigned long *ptr_qelsz;

d_bootact = BOOT_ACTIVE; /* show boot is active */
d_trapflg = i_ptr->trapflg; /* sysdebug state info */
d_mmu = i_ptr->mmu; /* mmu minimum block size */
d_intadd = i_ptr->intadd; /* internet address */
d_mputype = i_ptr->mputype; /* mpu type */
d_unimp_deb = i_ptr->unimp_deb; /* address of debuggers handlers */
d_incar = 0; /* initialize incarnation number */
d_init = i_ptr->init; /* pointer to initialization table */
d_boot = i_ptr->boot; /* base of boot jump table */
rom_list = i_ptr->rom_list; /* romlist pointer */

/* verify that memory management support is enabled */
if ( d_init->mmuen )
    {
    /* determine if there is hardware support */
    if ( d_mmu )
        {
        gblk(); /* get block of memory */
        }
    }
else
    {
    d_mmu = NO_MMU;
    }

/* set up private stack for exit system call */
d_exitstk = getram(EXIT_ST_SZ, EXIT_ST_CD) + EXIT_ST_SZ;

/* set up system call dispatch table */
d_sysdis = getram(d_init->scaltabsz, SYSCALL_CD);
/* obtain the end of the system dispatch table */
d_sysdise = d_sysdis + d_init->scaltabsz;

/* set up the module directory */

```



```

d_moddir = getram(d_init->mdsz, MODDIR_CD);
/* obtain the end of the module directory */
d_moddire = d_moddir + d_init->mdsz;

/* obtain the total process descriptor size */
tot_pdsz = d_init->numpd * PDSIZE;
/* set up the process descriptors */
d_pdba = getram(tot_pdsz, PD_CD);
/* obtain the end of the process descriptors */
d_pdbae = d_pdba + tot_pdsz;
/* obtain the head of the free process descriptor queue */
d_fprocq = d_pdba;

/* obtain the total path descriptor size */
tot_pathdsz = d_init->numptd * PATHSIZE;
/* set up the path descriptors */
d_pathba = getram(tot_pathdsz, PATHD_CD);
/* obtain the end of the path descriptors */
d_pathbae = d_pathba + tot_pathdsz;
/* obtain the head of the free path descriptor queue */
d_fpathq = d_pathba;

/* obtain the total device table size */
tot_devsz = d_init->numdev * DEVSIZE;
/* set up the device table */
d_devtbl = getram(tot_devsz, DEVTBL_CD);
/* obtain the end of the device table */
d_devtble = d_devtbl + tot_devsz;

/* set up the interrupt table */
d_irqtab = getram(IRQTABSZ, IRQTAB_CD);
/* obtain the end of the interrupt table */
d_irqtabe = d_irqtab + IRQTABSZ;

/* obtain the total mail box size */
tot_mailsz = d_init->postoff * MAILSIZE;
/* set up the head of the mail boxes */
d_postoff = getram(tot_mailsz, MAIL_CD);
/* obtain the end of the mail boxes */
d_postoffe = d_postoff + tot_mailsz;

/* obtain maximum of semaphores */
d_smnum = d_init->semnum;
/* obtain size per semaphore */
d_smsiz = d_init->numpd + SEMSZIE;
/* set up the semaphores */
d_smtbl = getram( d_smnum * d_smsiz, SEM_CD);

/* set up queue system */
ptr_numel = &(d_init->numqe);
/* point to the element size table */

```

```

ptr_qelsz = &(d_init->qelesz);

/* determine how much memory is needed for the queue system */
/* point to the number of queue elements table */
for ( que_mem = 0, i = 0; i < d_init->numques; i++)
    {
        no_que_el =*(ptr_num_el)++;
        sz_qele = *(ptr_qelsz)++;
        if (sz_qele % 2) /* test if even */
            sz_qele += EVEN_ALIGN;
        else
            sz_qele += ODD_ALIGN;
        que_sz = sz_qele * no_que_el;
        que_mem += que_sz;
    }

d_queue = getram(que_sz, QUE_CD);

/* obtain the pointer to free pointers used for enqueue */
d_queue->freeptr = d_queue + QUE_STRUCT_SZ;
/* set up pointer point to first element to dequeue from */
d_queue->dqptrf = d_queue->freeptr + d_init->numques;
/* set up pointer point to last element to dequeue from */
d_queue->dqptrl = d_queue->dqptrf + d_init->numques;
/* establish list of heads of processes waiting to enqueue */
d_queue->eqwait = d_queue->dqptrl + d_init->numques;
/* establish list of heads of processes waiting to dequeue */
d_queue->dqwait = d_queue->eqwait + d_init->numques;

que_ptr = d_queue->dqwait + d_init->numques;

/* point to number of elements per queue table */
ptr_numel = d_init->numqele;
/* point to the element size table */
ptr_qelsz = d_init->qelesz;

/* set up pointer to free elements in each queue */
for ( i = 0; i < d_init->numques; i++)
    {
        no_que_el =*(ptr_num_el)++;
        sz_qele = *(ptr_qelsz)++;
        if (sz_qele % 2) /* test if even */
            sz_qele += EVEN_ALIGN;
        else
            sz_qele += ODD_ALIGN;
        *(d_queue->freeptr) = que_ptr;
        que_ptr += ( sz_qele * no_que_el );
    }

/* point to number of elements per queue table */
ptr_numel = d_init->numqele;
/* point to the element size table */

```

```

ptr_qelsz = d_init->qelsz;

for ( i = 0; i < d_init->numques; i++)
    {
    no_que_el =*(ptr_num_el)++;
    free_elmnt = *(d_queue->free_ptr)++;
    sz_qele = *(ptr_qelsz)++;
    if (sz_qele % 2) /* test if even */
        sz_qele += EVEN_ALIGN;
    else
        sz_qele += ODD_ALIGN;

    /* set up the pointers to free elements */
    for ( j = 0; j < no_que_el; j++)
        {
        *free_elmnt = free_elmnt + sz_qele;
        free_elmnt = *free_elmnt;
        }
    *free_elmnt = NULL;
    }

/* The set up of mpu vector table is based upon the
existing value of the vector and the kernel vector image
table. The vector table is unaltered If there is a
leave vector in the image table if the original contents do
not point to the debugger's unimplemented exception
handler. */

/* get address of mpu vector table */
mpu_vec_ptr = MPU_VECTOR_ADD;
image_ptr = &vectab;

/* traverse through kernel image and the mpu vector tables */
for ( i = 0; i < VECTAB_SZ; i++, mpu_vec_ptr++, image_ptr++)
    {

    if ( (*mpu_vec_ptr != d_unimp_deb) ||
        (*image_ptr != LEAVEVEC) )

        {
        *mpu_vec_ptr = image_ptr + *image_ptr;
        }
    }

/* set up handlers for the following faults:
bus errors, address errors, illegal instruction, and trace */
for (err_index=0; err_index<MAX_INDEX; err_index++)
    {
    if (setvec(handler[err_index],vec_no[err_index]) == FAIL)
        {

```

```

        doom(err_msg[err_index], err_cd[err_index]);
    }
}

/* set up system and user dispatch tables */

/* search for modules */
/* Traverse through a list of blocks of memory searching for modules. This
list is a structure which consists of pointers to blocks of memory and
corresponding block sizes. */

for (block = rom_list; block->addr != NULL; block++)
{
    /* set up block limit */
    block_lim = block->addr + block->size;
    romptr = block->addr;

    if ( *romptr++ == M_IDVAL )
        { /* found sync word */
        --romptr;
        /* make sure header fits within the block */
        if ( romptr + (mod_hdr)rom_ptr->idsize <= block_lim )
            {
                /* verify that entire module fits within block */
                mod_size = (mod_hdr) romptr->mddsize;

                if ( romptr + mod_size <= block_lim )
                    /* run a parity check on the module */
                    && (prechk(romptr))
                    /* validate module */
                    && (validmod(romptr)) )
                    {

                        romptr = romptr + mod_size;
                        /* show module as rom */
                        (mod_hdr)romptr->attr |= ROM;
                    }
                else
                    {
                        romptr++;
                    }
            }
        }
}

/* link free process descriptors */

pdptr1 = pdptr2 = d_init->fprocq;

/* make this the first process descriptor */

```

```

pdptr1->pqueue = FAKEQHD( fprocq, nqueue);
pdptr1->pid = 1;

for ( i=0 ; i < d_init->numpd; i++)
    {
    eventinit();      /* initialize event variables */
    /* increment pointer by size of one process descr */
    pdptr1 += PDSIZE;
    /* increment process id */
    pdptr1->pid++;
    /* set state of process */
    pdptr->state |= DEAD;
    /* set current process to point to next */
    pdptr2->nqueue = pdptr1;
    /* current process is next pd's previous */
    pdptr1->pqueue = pdptr2;
    pdptr2 += PDSIZE;
    }

/* set up last pd */
pdptr1->nqueue = NULL;
/* obtain the address of the tick routine */
d_tckrtn = *(tick());

/* link free path descriptors */
pthdptr1 = d_init->fpathq;      /* get first path descriptor */

for ( i=0; i < d_init->numptd-1; i++)
    {
    /* point to next desc */
    pthdptr2 = pthdptr1 + PATHSIZE;
    /* link current path desc to next */
    pthdptr1->ptnqueue = pthdptr2;
    /* make next desc the current one */
    pthdptr1 = pthdptr2;
    }

/* mark the last free path in the list */
pthdptr->ptnqueue = NULL;

/* indicate that no longer booting */
d_bootact &= ^ACTIVE;

/* check for KEM (kernel extension modules) */
if ( findmod( d_init->kemstr, WILDCARD) )
    {
    /* kem exists, so set up process desc for kem chain */
    /* get a free process descriptor */
    d_procq = d_fprocq;
    /* fix free process descriptor pointers */
    d_fprocq = d_fprocq->nqueue;
    }

```

```

d_fprocq->pqueue = FAKEQHD(fprocq, nqueue);
/* fix current process desc */
d_procq->nqueue = NULL;

/* set up chain structure */
ch_st.module = d_init->kemstr;
ch_st.param = NOPARAM;
ch_st.typ_lang = PRGRM+M68KOBJ;
ch_st.opt = NO_OPT;
ch_st.parsz = 0;
ch_st.prior = DEF_PRI;
ch_st.int = DEF_IRQ;

/* kchain kem */
if ( kchain(ch_st) == -1)
    doom(sgerr, SG_CD);

/* kchain sysgo */
ch_st.module = d_init->sysgostr;
if ( kchain(ch_st) == -1)
    doom(sgerr, SG_CD);

}

else
{
/* get a free process descriptor */
d_procq = d_fprocq;
/* fix free process descriptor pointers */
d_fprocq = d_fprocq->nqueue;
d_fprocq->pqueue = FAKEQHD(fprocq, nqueue);
/* fix current process desc */
d_procq->nqueue = NULL;

/* set up chain structure */
ch_st.module = d_init->sysgostr;
ch_st.param = NOPARAM;
ch_st.typ_lang = PRGRM+M68KOBJ;
ch_st.opt = NO_OPT;
ch_st.parsz = 0;
ch_st.prior = DEF_PRI;
ch_st.int = DEF_IRQ;

/* chain sysgo */
if ( chain(ch_st) == -1)
    doom(sgerr, SG_CD);

}
}

```

```

/* iocall.c consists of the functions which handle all the I/O
   system services */

#include <io.h>
#include <errno.h>
#include <proc.h>

extern unsigned short io_code;
extern unsigned short dev_mode;
unsigned short path_no;

/* the create and open service call behave similarly. they both
   use the create_open function */

create(adrs_path, access_mode)
path_lst *adrs_path;
unsigned int access_mode;
{

io_code = I_CREATE;

return(create_open(adrs_path, access_mode));
}

open(adrs_path, access_mode)
path_lst *adrs_path;
unsigned int access_mode;
{

iocode = I_OPEN;

return(create_open(adrs_path, access_mode));
}

create_open(adrs_path, access_mode)
path_lst *adrs_path;
unsigned int access_mode;
{

path_d *data_ptr;
path_d *path_desc;

/* check if hardware path */
if (*adrs_path != PDELIM)
    {
        /* since not hardware path, check id using the default exec directory */
        if (access_mode & EXEC)
            {
                data_ptr = pd->dio; /* using default */
            }
    }
}

```

```

        }
    else
        {
            /* getting address of the device table entry */
            data_ptr = pd->device_tbl;
        }
    /* verify that path exists */
    if (data_ptr == 0)
        {
            errno = E_BPNam;
            return(FAIL);
        }
    }
else
    data_ptr = adrs_path;

/* attach device with each open in order to keep the link count accurate */
attach(data_ptr->desc, access_mode);

/* initialize a path descriptor */
pathinit(access_mode, data_ptr, &path_desc, &path_no);

/* claim ownership of the path */
getpath(path_no);

/* call file manager */
fmexec();

/* release path */
relpath(path_desc);

return(path_no);
}

/* make directory system service */
mkdir(adrs_path)
path_lst *adrs_path;
{

io_code = I_MAKDIR;
dev_mode = DIR_+WRITE;

mk_del(adrs_path);
}

/* delete system service */
delete(adrs_path)
path_lst *adrs_path;
{

```



```

io_code = I_DELETE;

mk_del(adrs_path);
}

// common setup code for mkdir and delete services */
mk_del(adrs_path)
path_lst *adrs_path;
{

unsigned short mode;
unsigned char def_flag; /* flag indicating that using default */
path_d *data_ptr;

/* check for hardware path */
if (*adrs_path != PDELIM)
{
/* not a hardware path, use default if it exists */
if (pd->dio != 0)
{
data_ptr = pd->dio;
def_flag = TRUE;
/* increment usage count */
data_ptr->v_usrst++;
}
else
{
/* does not exist, so error */
errno = E_BPNam;
return(FAIL);
}
}

else
{
/* hardware path */
def_flag = FALSE;
attach(adrs_path->desc, mode);
}

/* initialize path descriptor */
pathinit(mode, adrs_path, &path_desc, path_no);

/* invoke file manager */
fmexec();

/* return the path */
pathdinit(path_no);

if (def_flag == FALSE)

```

```

    {
    /* hardware path so detach device */
    detach(adrs_path->desc);
    }

else
    {
    /* decrement usage count */
    data_ptr->v_usr--;
    }
}

/* change directory system service */
chgdir(adrs_path, mode);
path_lst *adrs_path;
unsigned short mode;
{

unsigned char def_flag; /* flag indicating that using default */
path_d *data_ptr;

io_code = I_CHGDIR;

if (*adrs_path != PDELIM)
    {

        def_flag = TRUE;

        /* since not hardware path, check if using the default data directory */
        if (mode & UPDAT)
            {
                data_ptr = pd->dio; /* using default */
            }

        /* check if changing exec directory */
        else if (mode & EXEC)
            {
                /* getting address of the device table entry */
                data_ptr = pd->device_tbl;
            }

        else
            { /* error condition due to invalid mode */
                errno = E_BMode;
                return(FAIL);
            }

        /* verify that path exists */
        if (data_ptr == 0)
            {
                errno = E_BPNam;
            }
    }
}

```

```

        return(FAIL);
    }

    /* increment the use count */
    data_ptr->v_usrs--;
}

else
    { /* hardware path specified */
    def_flag = FALSE;
    attach(adrs_path->desc, mode);
    }

/* initialize path descriptor */
pathinit(mode, adrs_path, &path_desc, path_no);

/* invoke file manager */
fmexec();

/* return the path */
pathdinit(path_no);

if (def_flag = TRUE)
    {
    /* decrement use count */
    data_ptr->v_usrs--;
    }

/* determine which io ptr to replace */
else
    {
    if (pd->mode & EXEC)
        {
        /* replacing new default exec io ptr */
        pd->device_tbl = pd->dev;
        }

    else if (pd->mode & UPDAT)
        {
        /* replacing new default data io ptr */
        pd->dio = pd->dev;
        }

    /* hardware path so detach device */
    detach(adrs_path->desc);
    }
}

/* read system service call */
read(file_ptr, bytes_to_read)
path_d *file_ptr;

```

```

unsigned long bytes_to_read;
{

/* check memory to determine if valid memory */
chkmapwr(file_ptr);
io_code = I_READ;
return(io_common(file_ptr, bytes_to_read));
}

readln(file_ptr, bytes_to_read)
path_d *file_ptr;
unsigned long bytes_to_read;
{

chkmapwr(file_ptr);
io_code = I_READLN;
return(io_common(file_ptr, bytes_to_read));
}

/* write system call */
write(file_ptr, bytes_to_write)
path_d *file_ptr;
unsigned long bytes_to_write;
{

chkmaprd(file_ptr);           // validated the memory
io_code = I_WRITE;
return(io_common(file_ptr, bytes_to_write));
}

writeln(file_ptr, bytes_to_write)
path_d *file_ptr;
unsigned long bytes_to_write;
{

chkmaprd(file_ptr);           // validated the memory
io_code = I_WRITELN;
return(io_common(file_ptr, bytes_to_write));
}

/* getstat system service call */
getstat(path_ptr, stat_code)
path_d *path_ptr;
unsigned long stat_code;
{

io_code = I_GETSTT;
return(io_common(path_ptr, stat_code));
}

/* set stat system service call */

```

```

setstat(path_ptr, stat_code)
path_d *path_ptr;
unsigned long stat_code;
{

io_code = I_SETSTT;
io_common(path_ptr, stat_code);
}

/* display console system service */
discon(path_ptr, console_code)
path_d *path_ptr;
unsigned long console_code;
{

io_code = I_DISCON;
io_common(path_ptr, console_code);
}

/* common code pertaining to non open/close i/o system calls */
io_common(path_ptr, code)
path_d *path_ptr;
unsigned long code;
{

/* get device path */
getpath(path_no);

/* call file manager */
fmexec();

/* release path */
relpath(path_ptr);
}

/* duplicate i/o path system service */
dup(path_no)
{
int i;
path_d *path_desc;

/* verify that a valid path number was received */
if (path_no > NUM_PATHS-1)
    {
        errno = E_BPNUM;
        return(FAIL);
    }

else
    {

```

```

/* valid path number received, but need to check if there is a path
associated with the given path number */

path_desc = pd->path[path_no];

if (path_desc == 0)
    {
    errno = E_BPNUM;
    return(FAIL);
    }

else
    {
    /* search path for a free path */
    for (i=0, i<NUM_PATHS, i++)
        {
        if (pd->path[i] == 0)
            {
            /* found a free path, so copy path descriptor */
            path_desc->[i] = pd->path[path_no];
            /* increment use count */
            pd->pd.cnt++;
            return((unsigned short)i);
            }
        }
    /* did not find a free slot */
    errno = E_PNNF;
    return(FAIL);
    }
}

}

/* close system service call */
close()
{

path_d *path_desc;

/* verify that a valid path number was received */
if (path_no > NUM_PATHS-1)
    {
    errno = E_BPNUM;
    return(FAIL);
    }

else
    {
    /* valid path number received, but need to check if there is a path
associated with the given path number */

path_desc = pd->path[path_no];

```

```

    if (path_desc == 0)
        {
            errno = E_BPNUM;
            return(FAIL);
        }

    else
        {
            pd->pd.cnt--;
            /* get control of path */
            getpath(path_no);
            io_code = I_CLOSE;

            /* call file manager */
            fmexec();

            /* return path descriptor */
            retpath(path_no);
            detach(path_desc);
        }
    }
}

```

/\* attach will search the device table. If the device is not in the table it will attempt to add it if there are any free entries. attach will also call the device driver init routine and allocate any static storage the device driver may need \*/

```

attach( path_desc, access_mode)
path_d path_desc;
unsigned short access_mode;

{

int bytes_rcvd;
unsigned char *mem_ptr;
proc_desc *proc_ptr;
device_desc *dvc_drv;
file_manager *file_mgr;
unsigned long port_add;
unsigned long mod_entry;
unsigned long mod_hdr;

/* link to the device descriptor */
if (link( path_desc->desc, DEVIC, &mod_entry, &mod_hdr) == FAIL)
    {
        return(FAIL);
    }

dvc_drv = path_desc->desc->pdev;

```

```

/* link to device driver */
if (link(dvc_drv, DRIVR+M68KOBJ, &mod_entry, &mod_hdr) == FAIL)
    {
    /* problems linking to the device driver */
    /* unlink from device descriptor */
    unlink(path_desc->desc);
    return(FAIL);
    }

/* link to the file manager */
file_mgr = path_desc->desc->fmgr;

if (link(file_mgr, FLMGR+M68KOBJ, &mod_entry, &mod_hdr) == FAIL)
    {
    /* problems linking to the file manager */
    /* unlink from device descriptor & driver */
    unlink(path_desc->desc);
    unlink(dvc_drv);
    }

/* check access mode for compatibility with device descriptor
& driver */
if (((access_mode & path_desc->desc->mode) == 0)
    || ((access_mode & dvc_drv->mode) == 0)
    || ((access_mode & file_mgr->mode) == 0)
    {

    /* mismatch between access mode & device */
    errno = E_BTyp;

    /* unlink everything */
    unlink(path_desc->desc);
    unlink(dvc_drv);
    unlink(file_mgr);

    return(FAIL);
    }

/* search device table for device or attempt to install it in the table
if not present */
/* obtain the number of device entries from the init table */
dev_tbl_ent[0] = d_devtbl;
port_add = path_desc->desc->port;

for (i=0; i < d_init->numdev-1; i++)
    {
    /* check if file manager, device driver, & port address
are the same */
    if ((dev_tbl_ent[i]->fmgr == file_mgr)
        && (dev_tbl_ent[i]->pdev == dvc_drv)
        && (dev_tbl_ent[i]->port == port_add))

```



```

        {
        /* found device in the table */
        /* increment count of users */
        path_desc->v_usrst++;
        }
    }

/* did not find entry in the table, check for a free element */
dev_tbl_ent[0] = d_devtbl;

i=0;
while ((dev_tbl_ent[i]->desc != NULL) && (i<d_init->numdev-1))
    {
    i++;
    }

if (i >= d_init->numdev-1)
    {
    /* no free entries in the table */
    errno = E_DevOvf;

    /* unlink everything */
    unlink(path_desc->desc);
    unlink(dvc_drv);
    unlink(file_mgr);

    return(FAIL);
    }

else if (dev_tbl_ent[i]->desc == NULL)
    {
        /* get static storage for device driver */
        if ((dev_tbl_ent[i]->stat = extract(dvc_drv->mem, &bytes_rcvd)
            == NULL)
            {
            /* problems obtaining static storage for device driver */
            /* unlink everything */
            unlink(path_desc->desc);
            unlink(dvc_drv);
            unlink(file_mgr);

            return(FAIL);
            }

        mem_ptr = dev_tbl_ent[i]->stat;
        dvc_drv->mem = bytes_rcvd;

        /* initialize all static storage to zero */
        for (i=0; i<bytes_rcvd; i++)
            {
            *mem_ptr = 0;
            }
    }

```

```

    /* found a free entry */
    dev_tbl_ent[i]->desc = path_desc->desc;
    dev_tbl_ent[i]->pdev = dvc_drv;
    dev_tbl_ent[i]->fmgr = file_mgr;
    dev_tbl_ent[i]->port = port_add;

    /* mark device as being owned by process */
    file_mgr->busy = pd;
    /* set up port */
    file_mgr->port = port_add;

    /* invoke device driver initialization routine */
    dvc_drv->dinit();

    /* check if other process waiting to be service by file
       manager */
    proc_ptr = d_proc->fmnque;
    if (pd != NULL)
    {
        /* other process waiting on the file manager */
        /* remove process from list of waiting on file manager
           and activate the process */
        proc_ptr->fmpque = NULL;
        d_proc->fmnque = NULL;
        proc_ptr->state &= ^FMWAIT;
        proc_ptr->state |= ACTIVE;
        pd = proc_ptr;
        activateproc();
    }
}

/* detach removes the device from the device entry table */
detach(path_desc)
path_d *path_desc;
{
    dev_tbl_ent[0] = d_devtbl;

    for (i=0; i < d_init->numdev-1; i++)
    {
        if ((dev_tbl_ent[i]->fmgr == path_desc->desc->fmgr)
            && (dev_tbl_ent[i]->pdev == path_desc->desc->pdev)
            && (dev_tbl_ent[i]->port == path_desc->desc->port))
        {
            /* found device in the table */

```

```
/* unlink from the components of the i/o system */
unlink(path_desc->desc->fmgr);
unlink(path_desc->desc->pdev);
unlink(path_desc->desc);
```

```
/* if no other process using the same device so can
   deallocate the static storage and clear table entry */
```

```
if (path_desc->v_usr-- == 0)
```

```
{
```

```
liberate(dev_tbl_ent[i]->stat, path_desc->desc->pdev->mem);
```

```
dev_tbl_ent[i]->desc = NULL;
```

```
dev_tbl_ent[i]->fmgr = NULL;
```

```
dev_tbl_ent[i]->pdev = NULL;
```

```
dev_tbl_ent[i]->port = NULL;
```

```
}
```

```
}
```

```
}
```

```
}
```

```

/* iosubs.c      contains subroutines used by the io related system service
   requests. */

#include <errno.h>
#include <io.h>
#include <proc.h>

/* pathinit will acquire and initialize a path descriptor */

pathinit(access_mode, dev_tbl_adrs, *path_desc, *path_no)
unsigned short mode;
path_lst *dev_tbl_adrs;
path_d path_desc;
unsigned short path_no;
{

path_d *path_ptr;
path_d *path_tab;

path_ptr = pd->path;

/* search path table for free path */
for (i=0; i<NUMPATHS-1; i++)
    {

        if (path_ptr == NULL)
            {
                /* found a free path */
                path_no = i;

                /* assign free path pointer */
                path_ptr = allpath(path_no);

                /* set access mode */
                path_ptr->mode = access_mode;
                /* set address of device table entry */
                path_ptr->dev = dev_tbl_adrs;
                /* indicate that is being used */
                path_ptr->cnt++;

                /* put pointer to path in path table */
                path_tab = pd->path + path_no;
                *path_tab = path_ptr;

                /* copy options section */
                /* get number of bytes in options section */
                num_opts = dev_tbl_adrs->desc->opt;

                /* point to the first option */
                dev_tbl_adrs->desc->opt++;

                for (i=0; i<num_opts; i++)

```

```

        {
            path->opt++ = dev_tbl_adrs->desc->opt++;
        }
    }

    else
    {
        path_ptr++;
    }
}

errno = E_PNNF;
return(FAIL);
}

```

/\* pathdinit will remove entry from the path table and issue a call to return the path descriptor. \*/

```

pathdinit(path_no)
unsigned short path_no;

```

```

{

path_d *path_ptr;

/* verify that given path number is valid */
if (path_no > NUMPATHS-1)
    {
        errno = E_BPNum;
        return(FAIL);
    }

else
    {
        /* search the table for entry associated with given path number */
        path_ptr = pd->path;

        if ((path_ptr+path_no) != NULL)
            {
                /* path does exist, so return path descriptor */
                retpath(path_no);
            }

        else
            {
                errno = E_BPNum;
                return(FAIL);
            }
    }
}

```

```

/* getpath obtains a path */
getpath(path_no)
unsigned short path_no;
{
proc_desc *proc_ptr;
proc_desc *temp;
path_d *path_ptr;

/* validate the given path number */
if (path_no => NUNPATHS)
    {
    errno = E_BPNum;
    return(FAIL);
    }

else
    {
    /* check if path exists */
    path_ptr = pd->path;

    if ((path_ptr+path_no) != NULL)
        {
        /* path exists */
        path_ptr = path_ptr + path_no;

        /* check if path is free */
        if (path_ptr->cpr == NULL)
            {
            /* mark path as owned by current process */
            path_ptr->cpr = pd;
            /* so no queue */
            pd->auxpque = NULL;
            }

        else
            {
            /* currently being used */
            /* check if processes are waiting on this device */
            proc_ptr = path_ptr->cpr;
            if (proc_ptr->auxnque == NULL)
                {
                /* no other process waiting on device */
                proc_ptr->auxnque = pd;
                pd->auxpque = proc_ptr;
                pd->auxnque = NULL;
                }

            else
                {
                /* put into queue in accordance with priority */
                while ((proc_ptr->prior > pd->prior)
                    && (proc_ptr->auxnque != NULL))

```

```

        {
            /* point to next process in the queue */
            proc_ptr = proc_ptr->auxnque;
        }

        temp = proc_ptr->auxpque;
        temp->auxnque = pd;
        pd->pqueue = temp;
        pd->nqueue = proc_ptr;
        proc_ptr->pqueue = pd;
    }

    /* mark as waiting on a path */
    pd->state |= PATHWAIT;
    pd->state &= ^ACTIVE;
    nextproc();
    }
    }

else
    {
        errno = BPNUM;
        return(FAIL);
    }
}

```

```

/* relpath will release the path */
relpath(path_ptr)
path_d *path_ptr;
{

```

```

/* check if process waiting for path */
if((proc_ptr=pd->auxnque) != NULL)
    {
        /* there is a process waiting, so activate it */
        proc_ptr->auxpque = NULL;
        proc_ptr->state |= ACTIVE;
        proc_ptr->state &= ^PATHWAIT;
        activateproc();
        pd->auxnque = NULL;
    }

```

```

/* make previously waiting process the new owner of path */
path_ptr->cpr = proc_ptr;
}

```

```
/* mem.c contains the functions which handle memory related system
services */
```

```
/* srqmem handles the requests for more memory. this function relies
on the pcos debugger service, extract, for the actual allocation
of memory. */
```

```
#include <errno.h>
#include <proc.h>
```

```
srqmem(bytes_req, *ptr_block)
unsigned int bytes_req; /* bytes requested */
unsigned long ptr_block; /* pointer to granted block */
{
```

```
unsigned long bytes_granted;
```

```
/* check if supervisor mode request or a user request */
if (sysstate() == SUPERVISOR)
```

```
{
    ptr_block = extract(bytes_req, &bytes_granted);
    return(bytes_granted);
}
```

```
else
```

```
{ /* user mode */
    /* obtain memory block table pointer */
    mem_tbl->ptr = pd->memimg;
    mem_tbl_size = pd->blksiz;
```

```
/* traverse table searching for a free entry */
for (i=0; i<MEM_BLK_TBL_SZ; i++)
```

```
{
    if (mem_tbl->ptr[i] == 0)
    {
        /* found a free entry */
        ptr_block = extract(bytes_req, &bytes_granted);
        /* insert pointer into free entry */
        mem_tbl->ptr[i] = ptr_block;
        mem_tbl->size[i] = bytes_granted;
        return(bytes_granted);
    }
}
```

```
/* table is full */
errno = E_MemFul;
return(FAIL);
}
```

```
}
```

```
/* srtmem returns memory back to the free memory pool. this system service
```



```

uses the liberate, pcos debugger service. */

srtmem(bytes_returned, ptr_rtn_blk)
unsigned int bytes_returned;
unsigned long *ptr_rtn_blk;
{
/* check if supervisor mode request or a user request */
if (sysstate() == SUPERVISOR)
    {
    liberate(&bytes_returned, ptr_rtn_blk);
    return();
    }

else
    { /* user mode */

/* obtain memory block table pointer */
    mem_tbl->ptr = pd->memimg;
    mem_tbl->size = pd->blksiz;

    for (i=0; i<MEM_BLK_TBL_SZ; i++)
        {

/* traverse table searching for given pointer */
            if (ptr_rtn_blk == mem_tbl->ptr[i])
                {
/* found the given block pointer in the memory table */
                liberate(&bytes_returned, ptr_rtn_blk);
/* clear pointer and corresponding size */
                mem_tbl->ptr[i] = 0;
                mem_tbl->size[i] = 0;
                return;
                }
        }
    errno = E_BBPtr;
    return(FAIL);
    }
}

```

/\* setmem is responsible for setting the primary block of memory in memory table. \*/

```

setmem(req_mem_size, *granted_mem_size, *end_ptr)
unsigned int req_mem_size;
unsigned long *granted_mem_size;
unsigned long *end_ptr;
unsigned int bytes_granted;
unsigned long *ptr_block;
unsigned int bytes_returned;

{

```

```
/* verify that the requested memory is not zero. A zero would mean that
   the process wants to know the data size */
```

```
if (req_mem_size != 0)
{
    /* determine if requested size is less than current size */
    if (req_mem_size < pd->blksize[0])
    {
        /* verify that the user stack is safe */
        if (req_mem_size > pd->usp)
        {
            pd->blksize[0] = req_mem_size;
            ptr_excess = pd->meming[0] + req_mem_size;
            liberate(&bytes_returned, ptr_excess);
        }

        else
        {
            errno = E_DelSP;
            return(FAIL);
        }
    }

    else if (req_mem_size > pd->blksize[0])
    {
        /* requested size is greater than the current size */
        ptr_block = extract(req_mem_size, &bytes_granted);
        pd->blksize[0] = bytes_granted;
        pd->meming[0] = ptr_block;
    }
}

granted_mem_size = pd->blksize[0];
end_ptr = pd->meming[0] + pd->blksize[0];
}
```

```
/* module.c contains module-related services which are link, unlink,  
load, unload, findmod, and datmod */
```

```
#include <errno.h>  
#include <dir_page.h>  
#include <module.h>  
#include <proc.h>
```

```
extern unsigned short path_no;
```

```
/* link provides the service of linking to a memory module */
```

```
link(mod_ptr, type_lang, *mod_entry, *mod_hdr)
```

```
module *mod_ptr;
```

```
unsigned short type_lang;
```

```
unsigned long mod_entry;
```

```
unsigned long mod_hdr;
```

```
{
```

```
char *mod_name;
```

```
module *dir_addr;
```

```
mod_name = mod_ptr->name;
```

```
if (find_mod(mod_name, type_lang, &dir_addr) == PASSED)
```

```
{
```

```
/* check module attributes to determine if module
```

```
is re-entrant or if non-reentrant than check
```

```
if not being used */
```

```
if ((dir_addr->attr & REENTRANT) ||
```

```
(dir_addr->link == 0)
```

```
{
```

```
/* increment the link count */
```

```
dir_addr->link++;
```

```
mod_hdr = dir_addr;
```

```
mod_entry = dir_addr->exec;
```

```
/* need to determine if the module has read and write  
permissions */
```

```
if (dir_addr->attr & WATTR)
```

```
{
```

```
/* module has write attributes */
```

```
/* allocate segment with read & write permissions */
```

```
allseg(dir_addr->size, dir_addr->attr);
```

```
}
```

```
else
```

```
{
```

```
/* module has read only permission */
```

```
/* allocate segment with read only permissions */
```

```
allseg(dir_addr->size, dir_addr->attr|WRP);
```

```
}
```

```
}
```

```

        else
            {
                /* module is non-reentrant and is being used */
                errno = E_ModBsy;
                return(FAIL);
            }
    }

/* unlink is responsible for indicating that a module is not being
   used by the given process */
unlink(mod_hdr)
module *mod_hdr;
{

char *mod_name;
module *dir_addr;
unsigned short type_lang;

mod_name = mod_hdr->name;
type_lang = mod_hdr->type_lang;

/* verify that the given module exists */
if (findmod(mod_name, type_lang, &dir_addr) == PASSED)
    {
        /* decrement the link count associated with the module */
        dir_addr->link--;

        /* deallocate this module from caller */
        dallseg(dir_addr->size);
    }

else
    {
        return(FAIL);
    }

}

/* load function is responsible for loading a module into memory
   and adding it to the module directory */
load(path_ptr, access_mode)
{
unsigned long size;

/* open path */
path = open(path_ptr->name, access_mode);

```

```

/* get file size */
size = getstat(path_ptr, SS_SIZE);

/* extract memory */
mod_ptr = extract(size, &bytes_extracted);

/* read entire file */
read(mod_ptr, bytes_extracted);

/* verify that the entire file was loaded */
if ((got_eof = getstat(mod_ptr, SS_EOF)) == TRUE)
    {
        /* validate module */
        if (vmod(mod_ptr) == PASSED)
            {
                /* search module directory for an available entry */
                dir_entry = d_moddir;
                while (dir_entry <= d_moddire)
                    {
                        if (*dir_entry == NULL)
                            {
                                dir_entry = mod_ptr;
                                dir_entry->link = 0;
                                dir_entry->attr = mod_ptr->type_lang;
                            }
                    }

                errno = E_MDFull;
                return(FAIL);
            }
    }

else
    {
        /* did not load entire file */
        return(FAIL);
    }
}

/* unload function will remove a module from memory and delete
   its entry in the module directory */
unload(mod_hdr)
module *mod_hdr;
{
    md_entry *dir_entry;
    char *mod_name;

```

```

mod_name = mod_hdr->name;

/* search for module in the module directory */
/* verify that the given module exists */
if (findmod(mod_name, type_lang, &dir_entry) == PASSED)
    {
    /* check its link count and attributes */
    if (((mod_hdr->link == 0) && (mod_hdr->attr != STICKY))
        || ((mod_hdr->link == -1) && (mod_hdr->attr == STICKY))
        {

        /* clear its directory entry */
        dir_entry->link = 0;
        dir_entry->attr = 0;
        dir_entry = NULL;

        /* liberate its memory */
        liberate(mod_hdr, mod_hdr->size);
        }
    }

else
    {
    return(FAIL);
    }
}

```

```

/* findmod searches the module directory for the specified module */
findmod(mod_name, type_lang, *dir_entry)
char *mod_name;
unsigned short type_lang;
md_entry dir_entry;
{

dir_entry = d_moddir;

/* search the module directory from beginning to end */
while (dir_entry <= d_moddire)
    {
    if (dir_entry->mod_name == mod_name)
        {
        /* found with same name */
        /* verify that the type and language is the same */
        if (dir_entry->type_lang != type_lang)
            {
            type_lang = dir_entry->type_lang;
            }
        return(PASSED);
        }
    }

dir_entry += dir_entry->size;

```

```

    }

/* did not find the module in the module directory */
errno = E_MNF;
return(FAIL);

}

/* datmod function will create a data module */
datmod(mod_name, mod_size, attr, perm)
char *mod_name;
unsigned long mod_size;
unsigned short attr;
unsigned short perm;
{

if (mod_size == 0)
    {
    /* requested no memory */
    errno = E_ZMR;
    return(FAIL);
    }

/* the given size does not include header and crc */
/* add the size of the header and crc to mod_size */
mod_size += (HDR_SIZE + CRC_SIZE);

/* extract memory */
mod_ptr = extract(mod_size, &bytes_extracted);

/* clear allocated memory */
for (mem_ptr=mod_ptr, i=0; i<bytes_extracted; i++)
    {
    *mem_ptr = 0;
    }

/* build module header */
/* id */
mod_ptr->id = IDVAL;
/* size */
mod_ptr->size = bytes_extracted;
/* module ownership */
mod_ptr->owner = pd->user;
/* type language */
mod_ptr->type_lang = DATAMOD+M68KOBJ;
/* attributes and revision */
mod_ptr->attr = attr;
/* permission */
mod_ptr->accs = perm;
/* edition number */
mod_ptr->edition = 1;

```

```
/* name */
mod_ptr->name = mod_name;
/* base of data */
mod_ptr->dmbo = mod_ptr + HDR_SIZE;
/* in use flag */
mod_ptr->dmiuf = 0;

setcrc(mod_ptr);

/* validate module */
if (vmod(mod_ptr) == FAIL)
    {
    /* return memory */
    liberate(mod_ptr, bytes_extracted);
    return(FAIL);
    }
}
```



```

/* queues.c contains the service requests which deal with the queue
system. */

#include <errno.h>
#include <dir_page.h>
#include <proc.h>
#include <queue.h>

/* enqueue will place a queue element on a specific queue */
enqueue(que_num, elmn_sz, mode, *elmn_ptr)
unsigned long que_num;          /* queue number */
unsigned long elmn_sz;         /* element size */
unsigned short mode;           /* wait/no wait for free elements */
unsigned long *elmn_ptr; /* pointer to queue element to be enqueued */

{
proc_desc *qp_ptr;
proc_desc *ptr;
unsigned long free_elmn; /* number of elements that are free */

disable_irq();

/* verify that given queue number is valid */
if (d_init->numques < que_num)
    {
    errno = E_BADQN;
    return(FAIL);
    }

/* verify that there are free elements for given queue */
free_elmn = d_queue->freeptr + quenum;

/* are there any free elements? */
if (free_elmn == 0)
    {

    /* since no free element, check if should wait */
    if (mode == WAIT)
        {
        /* get current process */
        pd = d_procq;

        /* get pointer of process waiting to enqueue
to this queue */
        qp_ptr = d_queue->eqwait + que_num;

        /* check is someone is waiting for a free queue */
        if (*qp_ptr == NULL)
            {
            /* setup as head of queue since no one is waiting */
            pd->auxpqe = FAKEQHD(d_queue->eqwait, auxnque);
            }
        }
    }
}

```

```

        }

else
    {
        /* processes waiting to enqueue */
        /* find the end of the queue */
        for (ptr=qp_ptr; ptr->auxnque != NULL; )
            {
                ptr = ptr->auxnque;
            }
        /* put at the end of the queue */
        pd->auxpque = ptr;
    }
pd->auxnque = NULL;

/* show that process is not active and is waiting to enqueue */
pd->state &= ^ACTIVE;
pd->state |= QWAIT;
nextproc();
}

else
    {
        /* mode is no wait */
        errno = E_QFull;
        return(FAIL);
    }

/* was a kill signal received? */
if (pd->signal == KILL)
    {

        /* enqueue if there is a free element and a process wants to
        enqueue before termination */
        if (free_elmn && (*qp_ptr != NULL));
            {
                ptr = qp_ptr;
                if (ptr->auxnque != NULL)
                    {
                        qp_ptr = ptr->auxnque;
                        ptr->auxnque->auxpque = FAKEQHD(ptr->auxnque, pqueue);
                    }

                ptr->auxnque = NULL;
                ptr->auxpque = NULL;
                /* show process as active */
                ptr->state |= ACTIVE;
                /* process ready to enqueue */
                ptr->state &= QWAIT;
                activateproc(ptr);
            }
        exit(pd->signal);
    }

```

```

        }
    }

/* return pointer to free list */
free_elmn++;

/* get the size of the queue element for this queue */
qe_size = d_init->qelesiz + que_num;

if (chkmaprd(elmn_ptr, qe_size) == FAIL)
    {
        return(FAIL);
    }

/* check if the defined queue element size is the same
as the one trying to enqueue */
if (qe_size >= elmn_sz)
    {
        errno = E_BADQPTR;    /* bad queue pointer */
        return(FAIL);
    }

/* copy element data to queue */
for (i=0; i<elemn_sz; i++)
    {
        q_ptr[i] = *elmn_ptr++;
    }

/* get pointer of first queue element to dequeue */
dp_ptrf = d_queue->dqptrf + que_num;

/* get pointer of last queue element to dequeue */
dp_ptrl = d_queue->dqptrl + que_num;

/* check if dequeue list is empty */
if (*dq_ptrl == EMPTY)
    {
        /* first element on the dequeue list */
        *dp_ptrf = free_elmn;
        *dp_ptrl = free_elmn;
    }

else {
    /* not the first element on the list */
    *dp_ptrf = free_elmn;
}

/* null terminate the element list */
*free_elmn = NULL;

/* update queue status */
q_status = d_queue->qstat + que_num;

```

```

*q_stat++;

num_qel = d_init->numqele + que_num;
*num_qele -= *q_status;

/* get pointer of process waiting to dequeue from this queue */
q_ptr = d_queue->dqwait + que_num;

/* check if someone is waiting for a queue with data */
if (*q_ptr != NULL)
    {
    /* setup as head of queue since no one is waiting */
    ptr = *q_ptr;

    if (ptr->auxnque != NULL)
        {
        ptr->auxnque = FAKEQHD(ptr->auxnque, pqueue);
        }

    else
        {
        *q_ptr = NULL;
        }

    ptr->auxnque = NULL;
    ptr->auxpque = NULL;
    /* show process as active */
    ptr->state |= ACTIVE;
    /* process ready to enqueue */
    ptr->state &= ^QWAIT;
    activateproc(ptr);
    }
}

/* deque will get a queue element from a specified queue */
deque(que_num, elmn_sz, mode, *buf_ptr)
unsigned long que_num;          /* queue number */
unsigned long elmn_sz;         /* element size */
unsigned short mode;           /* wait/no wait for elements */
unsigned long *buf_ptr;        /* pointer to buffer where queue elements
                                are to be placed */

{
proc_desc *qp_ptr;
proc_desc *ptr;
unsigned long dq_elmn;         /* number of elements that are free */

disable_irq();

/* verify that given queue number is valid */
if (d_init->numques < que_num)

```

```

    {
    errno = E_BADQDN;
    return(FAIL);
    }

/* verify that there are elements in the specified queue */
dq_elmn = d_queue->dqptrf + quenum;

/* are there any elements to dequeue? */
if (!dq_elmn)
    {

    /* since no elements, check if should wait */
    if (mode == WAIT)
        {

        /* get current process */
        pd = d_procq;

        /* get pointer of process waiting to dequeue
           from this queue */
        q_ptr = d_queue->dqwait + que_num;

        /* check if someone is waiting for this queue */
        if (*q_ptr == NULL)
            {
            /* setup as head of queue since no one is waiting */
            pd->auxpque = FAKEQHD(d_queue->eqwait, auxnque);
            }

        else
            {
            /* process waiting to dequeue */
            /* find the end of the waiting queue */
            for (ptr=q_ptr, ptr->auxnque != NULL; )
                {
                ptr = ptr->auxnque;
                }
            /* put at the end of waiting queue */
            pd->auxpque = ptr;
            }

        pd->auxnque = NULL;

        /* show that process is not active and is waiting to dequeue */
        pd->state &= ^ACTIVE;
        pd->state |= QWAIT;
        nextproc();
        }

    else
        {

```

```

        /* mode is a no wait */
        errno = E_QEMPTY;
        return(FAIL);
    }

    /* was a kill signal received */
    if (pd->signal == KILL)
    {

        /* dequeue if there is an element to dequeue and a process
           wanting to dequeue before termination */
        if (dq_elmn && (*q_ptr != NULL))
        {
            ptr = q_ptr;

            if (ptr->auxnque != NULL)
            {
                q_ptr = ptr->auxnque;
                ptr->auxnque-auxpque = FAKEQHD(ptr->auxnque, pqueue);
            }

            ptr->auxnque = NULL;
            ptr->auxpque = NULL;
            /* show process as active */
            ptr->state |= ACTIVE;
            /* process ready to dequeue */
            ptr->state &= QWAIT;
            activateproc(ptr);
        }
        exit(pd->signal);
    }
}

/* queue is not empty */
dq_ptr++;

/* check if next is last */
if (*dq_ptr == NULL)
{
    /* mark it last */
    dq_ptr1 = NULL;
}

/* get the size of the queue element for this queue */
qe_size = d_init->qelesiz + que_num;

/* check if the defined queue element size is the same
   as the one trying to dequeue */
if (qe_size = d_init->qelesiz + que_num;
    {
        /* since actual < given, use actual */

```

```

    elmn_sz = que_size;
}

if (chkmaprd(buff_ptr, que_size) == FAIL)
{
    return(FAIL);
}

/* copy element from queue to buffer */
for (i=0; i<elmn_sz; i++)
{
    *buf_ptr++ = *dqptr++;
}

/* since finished dequeuing, have free elements */
free_elmn = d_queue->freeptr + que_num;
*free_elmn = dqptr;

/* update queue status */
q_status = d_queue->qstat + que_num;
*q_status--;

/* elements left in the queue */
eln_left = *q_status;

/* get pointer to list of processes waiting to enqueue to this queue */
q_ptr = d_queue->eqwait + que_num;

/* check if someone is waiting to enqueue */
if (*q_ptr != NULL)
{
    /* setup as head of queue since no one is waiting */
    ptr = *q_ptr;

    if (ptr->auxnque != NULL)
    {
        ptr->auxnque = FAKEQHD(ptr->auxnque, pqueue);
    }

    else
    {
        *q_ptr = NULL;
    }

    ptr->auxnque = NULL;
    ptr->auxpque = NULL;
    /* show process as active */
    ptr->state |= ACTIVE;
    /* process ready to enqueue */
    ptr->state &= ^QWAIT;
    activateproc(ptr);
}
}

```

```

/* qstat provides the status for a given queue */
qstat(que_num, que_avail, num_qele, elmn_sz, num_elmn)
unsigned long que_num;          /* queue number */
unsigned long *que_avail; /* total number of queues available */
unsigned long *num_qele;       /* number of elements per queue */
unsigned long *elmn_sz;       /* element size */
unsigned long *num_elmn;      /* number of elements in the queue */

{
/* verify that given queue number is valid */
if (d_init->numques < que_num)
    {
        errno = E_BADQON;
        return(FAIL);
    }

/* total number of queues on the system */
*que_avail = d_init->numques;

/* number of queue elements for this queue */
*num_qele = (unsigned long) d_init->numqele + que_num;

/* size of each element */
*elmn_sz = (unsigned long) d_init->qelesiz + que_num;

/* number of elements already in the queue */
*num_elmn = (unsigned long) d_queue->stat + que_num;
}

/* qclear will clear a given queue */
qclear(que_num)
unsigned long que_num;
{

/* verify that given queue number is valid */
if (d_init->numques < que_num)
    {
        errno = E_BADQON;
        return(FAIL);
    }

/* clear queue until there are no elements to dequeue */
que_ptr = d_queue->dqptrf;
free_elmn = d_queue->freeptrs + que_num;

while (que_ptr != NULL)
    {

```



```

if (*que_ptr == NULL)
    {
        que_ptr = NULL;
    }

*free_elmn = que_ptr;

/* update queue status */
q_status = d_queue->qstat + que_num;
*q_status--;

/* elements left in the queue */
elmn_left = *q_status;
}

/* get pointer of process waiting to enqueue to this queue */
q_ptr = d_queue->eqwait + que_num;

/* check if someone is waiting to enqueue */
if (*q_ptr != NULL)
    {

        /* setup as head of queue since no one is waiting */
        ptr = *q_ptr;

        if (ptr->auxnque != NULL)
            {
                ptr->auxnque->auxpque = FAKEQHD(ptr->auxnque, pqueue);
            }

        else
            {
                *q_ptr = NULL;
            }

        ptr->auxnque = NULL;
        ptr->auxpque = NULL;
        /* show process as active */
        ptr->state |= ACTIVE;
        /* process ready to enqueue */
        ptr->state &= ^QWAIT;
        activateproc(ptr);
    }
}

```

```

#include <errno.h>      /* errno defines */
#include <dir_page.h>   /* direct page header */
#include <proc.h>       /* process descriptor */
#include <time.h>       /* days in the month */

tick()
{

proc_desc *ptr_sleepq;

/* traverse through the sleep queue and activate any process
   that are done sleeping */
ptr_sleepq = d_sprocq;

while (ptr_sleepq != NULL)
    {
        if (d_stickcnt-- == 0)
            {
                /* tick count for the head of sleep process queue is zero,
                   check for other process which also have the same designated
                   sleep time */

                /* check if others finished sleeping */
                while (ptr_sleepq->sdelta == 0)
                    {
                        /* done sleeping activate process */
                        pd = ptr_sleepq;
                        activateproc();
                        ptr_sleepq = ptr_sleep->nqueue;
                    }

                /* set up new queue head & re-initialize lowest sleep tick count */
                d_sproc = ptr_sleepq;
                d_stickcnt = ptr_sleepq->sdelta;
            }
    }

/* if there are any timers, check if they have timed out */
if (d_timptr != NULL)
    {
        /* timers have been set */
        /* decrement the head timer tick count */
        if (d_cticks-- == 0)
            {
                /* timeout */
                timeout();
            }
    }

/* update counters */

```

```

d_daytick++;    /* increment count ticks per day */

/* has the count of ticks per second reached zero */
if (d_tick-- != 0)
{
    /* re-initialize tick with ticks per second */
    d_tick = d_tcksec;

    /* decrement seconds per count, check is day passed */
    if (d_second-- != 0)
    {
        /* update seconds in a day */
        d_second = SEC_PER_DAY;
        /* re-initialize ticks per day */
        d_daytick = 0;
        /* update counts for julian days and days */
        d_julian++;
        d_day++;

        /* decrement days per month count, check if month passed */
        if (d_daynot-- == 0)
        {
            /* re-initialize count of days */
            d_day = 1;

            if (d_month++ > 12)
            {
                /* wrap around */
                d_month -= 12;
                /* year has elapsed */
                d_year++;
            }

            /* re-initialize days per month */
            d_daynot = month_days[d_month];

            /* check if leap year and month of February */
            if ((d_month == FEB) &&
                (((d_year % 4) == 0) && ((d_year % 100) != 0))
                || (d_year % 400) == 0)
            {
                /* February has 1 more day */
                d_daynot++;
            }
        }
    }
}

/* if no current process bypass record keeping */
if ((pd = d_proc) != 0)

```

```

    {
    if (pd->state == SYSTEM_STATE)
        {
        pd->sticks++; /* increment number of system ticks */
        }
    else
        {
        pd->uticks++; /* increment number of user ticks */
        }
    }

/* decrement ticks per slice */
if (d_slice-- == 0)
    {
    /* time slice expired */
    /* re-initialize time slice */
    d_slice = d_tslice;

    /* check if current process is in locked mode */
    if (pd->lstate != LOCKED)
        {
        /* verify that current process is not the only process that can be
        active */
        if (d_aprocq != 0)
            {
            /* indicate that time expired for current process */
            pd->state = TIMEOUT;
            /* sort the queue for active processes */
            activateproc();
            /* let the next active process in the queue run */
            nextproc();
            }
        }
    }
}

```

```

/* sort the queue of active processes */
activateproc()
{

proc_desc *proc_ptr;
proc_desc *temp;

/* disable the interrupts */
disable_irq();

/* preserve state relevant info & mark state as active */
pd->state &= STCPRESERVE;

```

```

pd->state |= ACTIVE;

/* set age equal to priority */
pd->age = pd->prior;

/* check is the active process queue is empty */
if (d_aprocq == 0)
{
    /* since queue empty, place process at the head of queue */
    d_aprocq = pd;
    pd->nqueue = NUL;
    pd->pqueue = FAKEQHD(d_aprocq, pqueue);
    return;
}

else
{
    /* active process queue is not empty, so need to sort */
    proc_ptr = d_aprocq; /* point to the top of the queue */

    /* age all processes in the queue, do not exceed 0xffff */
    while (proc_ptr->nqueue != NULL)
    {
        if (proc_ptr->age < 0xffff)
        {
            proc_ptr->age++;
        }
        proc_ptr = proc_ptr->nqueue;
    }

    /* point again to the top of the queue */
    proc_ptr = d_aprocq;

    while (proc_ptr->nqueue != NULL)
    {
        /* check if new process has a greater priority than process
        in queue */
        if (pd->age < proc_ptr->age)
        {
            /* new process has a lower priority */
            proc_ptr = proc_ptr->nqueue;
        }

        else
        {
            /* insert new process into queue according to its priority */
            temp = proc_ptr->pqueue;
            temp->nqueue = pd;
            pd->pqueue = temp;
            pd->nqueue = proc_ptr;
            proc_ptr->pqueue = pd;
        }
    }
}

```

```

        }
    }
}

/* the process at the head of the active process queue is given
   the processor. nextproc is responsible for dispatching. */
nextproc()
{
    disable_irq();

    /* point to current process and save its context*/
    pd = d_proc;
    save_regs(pd);

    /* the current process is now the head of the active
       process queue */
    d_proc = d_aprocq;

    /* check if mmu is enabled */
    if (d_mmu)
    {
        /* if mmu is enabled, activate segments */
        *boot[actseg]();
    }

    /* set up new head of the active process queue */
    d_aprocq = d_proc->nqueue;

    d_aprocq->pqueue = FAKEQHD(d_aprocq, pqueue);

    /* the current process is no longer tied to active process queue */
    d_proc->nqueue = NULL;
    d_proc->pqueue = NULL;

    /* check if signal is pending */
    if (d_proc->signal)
    {
        /* received signal */
        /* verify that signal is not a KILL signal & determine if a
           signal handler exists for the pending signal */
        if ((d_proc->signal == KILL) ||
            (d_proc->sigvec == NULL))
        {
            exit(d_proc->signal);
        }
    }
    else

```

```

        {
        /* setup process to handle signal */
        process_signal();
        }
    }

/* initialize slice to ticks per time slice */
d_slice = d_tslice;

/* restore context */
restore_regs(d_proc);

}

/* lock will not allow the kernel to give control of the processor
   to another process for a given period of time */
lock(no_tcks)
unsigned short no_tcks;
{

/* verify that given a valid time period */
if (no_tcks > 0)
    {
        /* set the new tick slice count */
        d_slice = no_tcks;

        /* show in a locked state */
        pd->lstate |= LOCKED;
        pd->state &= ^TIMOUT;
    }

else
    {
        errno = E_BadCnt;
        return(FAIL);
    }
}

/* unlock will remove the lock */
unlock()
{

disable_irq();

/* state will now reflect unlock mode */
pd->state &= ^LOCKED;

/* check if lock timed out */

```

```
if (pd->lstate | LOCKTO)
{
    /* lock has not timed out */
    /* re-initialize time slice */
    d_slice = d_tslice;
}

else
{
    /* lock had already expired */
    errno = E_LockExp;
    pd->lstate &= ^LOCKTO;
    return(FAIL);
}
}
```



```

#include <errno.h>      /* errno defines */
#include <dir_page.h>   /* direct page header */
#include <proc.h>       /* process descriptor */
#include <time.h>       /* days in the month */

tick()
{

proc_desc *ptr_sleepq;

/* traverse through the sleep queue and activate any process
   that are done sleeping */
ptr_sleepq = d_sprocq;

while (ptr_sleepq != NULL)
    {
    if (d_stickcnt-- == 0)
        {
        /* tick count for the head of sleep process queue is zero,
           check for other process which also have the same designated
           sleep time */

        /* check if others finished sleeping */
        while (ptr_sleepq->sdelta == 0)
            {
            /* done sleeping activate process */
            pd = ptr_sleepq;
            activateproc();
            ptr_sleepq = ptr_sleep->nqueue;
            }

        /* set up new queue head & re-initialize lowest sleep tick count */
        d_sproc = ptr_sleepq;
        d_stickcnt = ptr_sleepq->sdelta;
        }
    }

/* if there are any timers, check if they have timed out */
if (d_timprr != NULL)
    {
    /* timers have been set */
    /* decrement the head timer tick count */
    if (d_cticks-- == 0)
        {
        /* timeout */
        timeout();
        }
    }

/* update counters */

```

```

d_daytick++;    /* increment count ticks per day */

/* has the count of ticks per second reached zero */
if (d_tick-- != 0)
{
    /* re-initialize tick with ticks per second */
    d_tick = d_tcksec;

    /* decrement seconds per count, check is day passed */
    if (d_second-- != 0)
    {
        /* update seconds in a day */
        d_second = SEC_PER_DAY;
        /* re-initialize ticks per day */
        d_daytick = 0;
        /* update counts for julian days and days */
        d_julian++;
        d_day++;

        /* decrement days per month count, check if month passed */
        if (d_daynot-- == 0)
        {
            /* re-initialize count of days */
            d_day = 1;

            if (d_month++ > 12)
            {
                /* wrap around */
                d_month -= 12;
                /* year has elapsed */
                d_year++;
            }

            /* re-initialize days per month */
            d_daynot = month_days[d_month];

            /* check if leap year and month of February */
            if ((d_month == FEB) &&
                (((d_year % 4) == 0) && ((d_year % 100) != 0))
                || (d_year % 400) == 0))
            {
                /* February has 1 more day */
                d_daynot++;
            }
        }
    }
}

/* if no current process bypass record keeping */
if ((pd = d_proc) != 0)

```

```

    {
    if (pd->state == SYSTEM_STATE)
        {
            pd->sticks++; /* increment number of system ticks */
        }
    else
        {
            pd->uticks++; /* increment number of user ticks */
        }
    }

/* decrement ticks per slice */
if (d_slice-- == 0)
    {
        /* time slice expired */
        /* re-initialize time slice */
        d_slice = d_tslice;

        /* check if current process is in locked mode */
        if (pd->lstate != LOCKED)
            {
                /* verify that current process is not the only process that can be
                active */
                if (d_aprocq != 0)
                    {
                        /* indicate that time expired for current process */
                        pd->state = TIMEOUT;
                        /* sort the queue for active processes */
                        activateproc();
                        /* let the next active process in the queue run */
                        nextproc();
                    }
            }
    }
}

```

```

/* sort the queue of active processes */
activateproc()
{

proc_desc *proc_ptr;
proc_desc *temp;

/* disable the interrupts */
disable_irq();

/* preserve state relevant info & mark state as active */
pd->state &= STCPRESERVE;

```

```

    {
    if (pd->state == SYSTEM_STATE)
        {
        pd->sticks++; /* increment number of system ticks */
        }
    else
        {
        pd->uticks++; /* increment number of user ticks */
        }
    }

/* decrement ticks per slice */
if (d_slice-- == 0)
    {
    /* time slice expired */
    /* re-initialize time slice */
    d_slice = d_tslice;

    /* check if current process is in locked mode */
    if (pd->lstate != LOCKED)
        {
        /* verify that current process is not the only process that can be
        active */
        if (d_aprocq != 0)
            {
            /* indicate that time expired for current process */
            pd->state = TIMEOUT;
            /* sort the queue for active processes */
            activateproc();
            /* let the next active process in the queue run */
            nextproc();
            }
        }
    }
}

```

```

/* sort the queue of active processes */
activateproc()
{

proc_desc *proc_ptr;
proc_desc *temp;

/* disable the interrupts */
disable_irq();

/* preserve state relevant info & mark state as active */
pd->state &= STCPRESERVE;

```

```

pd->state |= ACTIVE;

/* set age equal to priority */
pd->age = pd->prior;

/* check is the active process queue is empty */
if (d_aprocq == 0)
    {
    /* since queue empty, place process at the head of queue */
    d_aprocq = pd;
    pd->nqueue = NUL;
    pd->pqueue = FAKEQHD(d_aprocq, pqueue);
    return;
    }

else
    {
    /* active process queue is not empty, so need to sort */
    proc_ptr = d_aprocq; /* point to the top of the queue */

    /* age all processes in the queue, do not exceed 0xffff */
    while (proc_ptr->nqueue != NULL)
        {
        if (proc_ptr->age < 0xffff)
            {
            proc_ptr->age++;
            }
        proc_ptr = proc_ptr->nqueue;
        }

    /* point again to the top of the queue */
    proc_ptr = d_aprocq;

    while (proc_ptr->nqueue != NULL)
        {

        /* check if new process has a greater priority than process
        in queue */
        if (pd->age < proc_ptr->age)
            {
            /* new process has a lower priority */
            proc_ptr = proc_ptr->nqueue;
            }

        else
            {
            /* insert new process into queue according to its priority */
            temp = proc_ptr->pqueue;
            temp->nqueue = pd;
            pd->pqueue = temp;
            pd->nqueue = proc_ptr;
            proc_ptr->pqueue = pd;
            }
        }
    }

```

```

        }
    }
}

/* the process at the head of the active process queue is given
   the processor. nextproc is responsible for dispatching. */
nextproc()
{
    disable_irq();

    /* point to current process and save its context*/
    pd = d_proc;
    save_regs(pd);

    /* the current process is now the head of the active
       process queue */
    d_proc = d_aprocq;

    /* check if mmu is enabled */
    if (d_mmu)
    {
        /* if mmu is enabled, activate segments */
        *boot[actseg]();
    }

    /* set up new head of the active process queue */
    d_aprocq = d_proc->nqueue;

    d_aprocq->pqueue = FAKEQHD(d_aprocq, pqueue);

    /* the current process is no longer tied to active process queue */
    d_proc->nqueue = NULL;
    d_proc->pqueue = NULL;

    /* check if signal is pending */
    if (d_proc->signal)
    {
        /* received signal */
        /* verify that signal is not a KILL signal & determine if a
           signal handler exists for the pending signal */
        if ((d_proc->signal == KILL) ||
            (d_proc->sigvec == NULL))
        {
            exit(d_proc->signal);
        }
    }
    else

```

```

        {
        /* setup process to handle signal */
        process_signal();
        }
    }

/* initialize slice to ticks per time slice */
d_slice = d_tslice;

/* restore context */
restore_regs(d_proc);

}

/* lock will not allow the kernel to give control of the processor
to another process for a given period of time */
lock(no_ticks)
unsigned short no_ticks;
{

/* verify that given a valid time period */
if (no_ticks > 0)
    {
    /* set the new tick slice count */
    d_slice = no_ticks;

    /* show in a locked state */
    pd->lstate |= LOCKED;
    pd->state &= ^TIMOUT;
    }

else
    {
    errno = E_BadCnt;
    return(FAIL);
    }
}

/* unlock will remove the lock */
unlock()
{

disable_irq();

/* state will now reflect unlock mode */
pd->state &= ^LOCKED;

/* check if lock timed out */

```

```
if (pd->lstate | LOCKTO)
{
    /* lock has not timed out */
    /* re-initialize time slice */
    d_slice = d_tslice;
}

else
{
    /* lock had already expired */
    errno = E_LockExp;
    pd->lstate &= ^LOCKTO;
    return(FAIL);
}
}
```



```
/* time.c contains the functions which either set the system time and
   date or return the system time and date */
```

```
#include <dir_page.h>
#include <errno.h>
#include <time.h>
```

```
/* the setime function will set the system time and date */
setime(time, date)
unsigned long time;          /* format 00hhmmss */
unsigned long date;        /* format yyymmdd */
{
```

```
    unsigned long year;
    unsigned short month;
    unsigned char day;
    unsigned char seconds;
    unsigned char minutes;
    unsigned char hours;
    unsigned long julian_time;
    unsigned long mod_entry;
    unsigned long mod_hdr;
```

```
    /* setting the date */
    /* strip year out of the given date */
    year = date;
    year = year >> 4;
```

```
    d_year = year;
```

```
    /* strip month out from the given date */
    month = date >> 2;
    month &= MONTHMASK;
```

```
    /* verify that received a valid month input */
    if ((month < 1) || (month > 12))
    {
        errno = E_Setime;
        return(FAIL);
    }
```

```
    else
```

```
    {
        /* received a valid month designator */
        /* verify that given day is valid for
           the given month */
        day = (unsigned char) date;
```

```
        if ((day < 1) || ((month != FEB) && (day > month_days[month])))
        {
```

```

        errno = E_Setime;
        return(FAIL);
    }

    if (month == FEB)
    {
        /* check if it is a leap year */
        if (((year % 4) == 0) && ((year % 100) != 0))
            || ((year % 400) == 0))
        {

            /* it is a leap year */
            feb_days = month_days[FEB] + 1;
        }
        else
        { /* not a leap year */
            feb_days = month_days[FEB];
        }
    }

    if (day > feb_days)
    {
        errno = E_Setime;
        return(FAIL);
    }
    d_day = day;
    d_daynot = month_days[month] - d_day;
}

/* setting the time */
seconds = (unsigned char) time;
minutes = (unsigned char) (time >> 2);
hours = (unsigned char) (time >> 4);

/* verify that the given time is valid */
if ((seconds > 59) || (minutes > 59) || (hours > 23))
{
    errno = E_Setime;
    return(FAIL);
}

/* get clock module */
if (d_init->clockstr == NULL)
{
    /* could the pointer to the clock module */
    errno = E_NoClk;
    return(FAIL);
}

else
{
    /* link to the clock module & execute its initialization routine */

```

```

link(d_init->clockstr, SYSTEM+M68KOBJ, &mod_entry,
     &mod_hdr);

*clock_rtn = mod_entry->exec;

disable_irq();

clock_rtn();

/* convert to julian date & time */
/* fjulian function returns julian time which corresponds to
   seconds since midnight */
julian_time = fjulian(time, date);
/* obtain seconds until midnight */
d_second = ^julian_time + SEC_PER_DAY;

/* calculate current tick in day */
d_daytick = (unsigned long)d_tcksec * d_second;
}
}

/* fjulian converts gregorian date to julian date & returns julian time */
fjulian( time, date)
unsigned long time;
unsigned long date;
{
    unsigned long jul_time;
    unsigned long day;
    unsigned long month;
    unsigned long year;

    unsigned long yr_fact; /* year factor associated with post 1582 */
    long yr_div; /* year divided by 100 */
    unsigned long jul_yr; /* julian year */
    unsigned long jul_mon; /* julian month */

    unsigned long hours;
    unsigned long hr_in_min; /* hours converted to minutes */
    unsigned long minutes;
    unsigned long min_in_sec; /* minutes converted to seconds */
    unsigned long min_sum; /* tally of minutes */
    unsigned long seconds;

    /* strip day, month, & year from the given date */
    day = date & DAY_MASK;
    month = (date >> 2) & MONTH_MASK;
    year = (date >> 4) & YEAR_MASK;

    if (month < 3)

```

```

        {
        year--;
        month += 12;
        }

yr_fact = 0;

if ((year > 1582) || ((year == 1582) && (month > 10))
    || ((year == 1582) && (month == 10) && (day > 15))
    {
    yr_div = year/100;
    yr_fact = 2 - yr_div + (yr_div/4);
    }

jul_yr = (year * 365.25) + yr_fact;

month++;
jul_mon = 30.6001 * month;

/* set the julian date */
d_julian = jul_yr + jul_mon + day + JUL_CONST;

/* convert from hhmms to seconds */
/* convert the hours to minutes */
hours = (time >> 4) & HR_MASK;
hr_in_min = hours * 60;

/* get total amount of minutes */
minutes = (time >> 2) & MIN_MASK;
min_sum = hr_in_min + minutes;

/* convert minutes to seconds */
min_in_sec = min_sum * 60;
seconds = time & SEC_MASK;

/* julian time is in seconds */
jul_time = seconds + min_in_sec;

return(jul_time);
}

/* rettime returns the system date and time. The user selects the
   desired format either gregorian or julian */
rettime( format, *time, *date)
unsigned short format;
unsigned long time;
unsigned long date;
{

```

```

unsigned long current_sec;
unsigned long hour;
unsigned long minute;
unsigned long second;

/* get seconds since midnight */
current_sec = SEC_PER_DAY - d_second;

if (format == JULIAN)
    {
    time = current_sec;
    date = d_julian;
    }

else
    {
    /* gregorian format */
    /* time format: 00hhmmss */
    hour = current_sec / SEC_PER_HR;

    minute = (current_sec - (hour * SEC_PER_HR))/SEC_PER_MIN;

    second = current_sec - (minute * SEC_PER_MIN);

    time = hour << 4;
    time |= (minute << 2);
    time |= second;

    /* date format: yyyyymmdd */
    date = d_year << 4;
    date |= (d_month << 2);
    date |= d_day;
    }
}

```

```
/* timer.c contains functions which handle timers. These function
   include timeout, clrtimer, and setimer. */
```

```
#include <errno.h>
#include <timer.h>
#include <dir_page.h>
```

```
/* timeout handles the processing of any timeouts */
timeout()
{
```

```
    tmr_struct *temp;
    tmr_struct *next_tmr;
```

```
    prt_tmr = d_timptr;
```

```
    /* traverse the table, checking for any timeouts. delta time of zero
       indicates a timeout */
```

```
    while (ptr_tmr->delta == 0)
    {
```

```
        /* timeout, so send event to corresponding process */
        sendevent(ptr_tmr->id, ptr_tmr->incarn, ptr_tmr->opcode,
                  ptr_tmr->address);
```

```
        /* check if periodic or one shot timer */
```

```
        if (ptr_tmr->mode != PERIODIC)
```

```
        {
            /* one shot timer so remove from time table */
            clrtimer(ptr_tmr->tnumber);
        }
```

```
    else
```

```
        { /* periodic, so reinitialize period */
            ptr_tmr->delta = ptr_tmr->period;
        }
```

```
        ptr_tmr = ptr_tmr->next;
    }
```

```
/* get top of the timer linked list */
```

```
prt_tmr = d_timptr;
```

```
/* sort timer table in increasing delta time */
```

```
while (ptr_tmr->next != NULL)
```

```
{
    next_tmr = ptr_tmr->next;
```

```
    if (ptr_tmr->delta > next_tmr->delta)
```

```
    {
        temp = ptr_tmr;
```

```

        ptr_tmr = ptr_tmr->next;
        ptr_tmr->next = temp;
    }
}

/* since cticks is zero, set it to lowest delta value
   and adjust all deltas accordingly */
d_cticks = d_timptr->delta;

while (ptr_tmr->next != NULL)
    {
        ptr_tmr->delta = ptr_tmr->delta - d_cticks;
    }
}

/* clrtimer clears the timer entry in the timer table for a given timer
   number */

clrtimer(timer_no)
int timer_no;

{

tmr_struct *prev_tmr;

/* verify that the given timer is valid */
if ((timer_no < 0) || timer_no > d_no_tmr)
    {
        errno = E_BTmrNo;
        return(FAIL);
    }

else
    {
        ptr_tmr = d_timptr;

        /* determine if there is a entry associated with given timer */
        while (ptr_tmr->next != NULL)
            {
                if (ptr_tmr->tnumber == timer_no)
                    {
                        /* found corresponding timer, so remove */
                        prev_tmr = ptr_tmr->prev;
                        prev_tmr->next = ptr_tmr->next;
                        return;
                    }
                ptr_tmr = ptr_tmr->next;
            }
        errno = E_noTimer;
        return(FAIL);
    }
}

```

```

    }
}

/* setimer will add a timer to the list. The timer is inserted in the list
   in accordance with its period. */
setimer(period, pid, pincarn, opcode, address)
unsigned long period;
unsigned short pid;
unsigned long pincarn;
unsigned long opcode;
unsigned long address;

{

tmr_struct *temp;
tmr_struct *prev_tmr;

/* verify that have not exceed the limit on the number of timers */
if (d_no_tmr+1 <= MAX_TIMERS)
    {
    ptr_tmr = d_timptr;

        /* determine where to insert new timer */
        while (ptr_tmr->next != NULL)
            {
            if ((period - d_cticks) > ptr_tmr->delta)
                {
                ptr_tmr = ptr_tmr->next;
                }

            else
                {
                /* insert timer into the list */
                prev_tmr = ptr_tmr->prev;
                prev_tmr->next = temp;
                temp->next = ptr_tmr;
                ptr_tmr->prev = temp;
                temp->prev = prev_tmr;

                /* set up structure with given information */
                temp->id = pid;
                temp->incarn = pincarn;
                temp->opcode = opcode;
                temp->address = address;
                temp->period = period;

                temp->delta = period - d_cticks;

                /* get the largest timer number and increment it */
                d_no_tmr += 1;
                timer_no = d_no_tmr;
                }
            }
    }
}

```



```
        temp->tnumber = timer_no;
        return(timer_no);
    }
else
    {
    errno = E_NoTMR;
    return(FAIL);
    }
}
```