INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
SPENCER, ROBERT DOUGLAS

A REAL-TIME MULTI-TASKING OPERATING SYSTEM FOR MICROCOMPUTERS

THE UNIVERSITY OF ARIZONA

M.S. 1984

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106
A REAL-TIME MULTI-TASKING OPERATING SYSTEM FOR MICROCOMPUTERS

by

Robert Douglas Spencer

A Thesis Submitted to the Faculty of the DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING In Partial Fulfillment of the Requirements For the Degree of MASTER OF SCIENCE WITH A MAJOR IN ELECTRICAL ENGINEERING In the Graduate College THE UNIVERSITY OF ARIZONA

1984
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: [Signature]

APPROVAL BY THE THESIS DIRECTOR

This thesis has been approved on the date shown below:

T. L. Williams

T. L. Williams
Professor of Electrical Engineering

August 30, 1984

Date:
ACKNOWLEDGMENTS

First, I would like to thank the University of Arizona department of Electrical and Computer Engineering for offering advance degrees to students whose bachelors degrees were not in this field of study. If the non-major's graduate program had not been available, I would not have been able to pursue the master's degree.

Second, I would like to extend my thanks to Dr. T. L. Williams whose guidance and instruction have provided me with a strong foundation on which to continue work in the field of computer system design.

Finally, I would like to express my deepest appreciation for the support and patience given me by my wife and family.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Concepts</td>
<td>2</td>
</tr>
<tr>
<td>2. HARDWARE BASE</td>
<td>6</td>
</tr>
<tr>
<td>MC68000</td>
<td>7</td>
</tr>
<tr>
<td>Memory Map and Management</td>
<td>8</td>
</tr>
<tr>
<td>3. THE SYSTEM</td>
<td>10</td>
</tr>
<tr>
<td>Console Monitor</td>
<td>10</td>
</tr>
<tr>
<td>Error Handling</td>
<td>12</td>
</tr>
<tr>
<td>Booting QUAD4</td>
<td>13</td>
</tr>
<tr>
<td>Operating System</td>
<td>13</td>
</tr>
<tr>
<td>File I/O</td>
<td>15</td>
</tr>
<tr>
<td>I/O Redirection</td>
<td>17</td>
</tr>
<tr>
<td>Multiple Commands</td>
<td>18</td>
</tr>
<tr>
<td>A New Run</td>
<td>19</td>
</tr>
<tr>
<td>In-Line Files</td>
<td>20</td>
</tr>
<tr>
<td>Special Characters</td>
<td>21</td>
</tr>
<tr>
<td>System Calls</td>
<td>21</td>
</tr>
<tr>
<td>4. THE MEMORY MANAGER</td>
<td>23</td>
</tr>
<tr>
<td>Bit Map Allocation</td>
<td>23</td>
</tr>
<tr>
<td>Alloc and Free</td>
<td>23</td>
</tr>
<tr>
<td>Core Allocation</td>
<td>24</td>
</tr>
<tr>
<td>Memory Modules</td>
<td>25</td>
</tr>
<tr>
<td>Relocation</td>
<td>26</td>
</tr>
<tr>
<td>5. THE I/O MANAGER</td>
<td>28</td>
</tr>
<tr>
<td>Connect</td>
<td>29</td>
</tr>
<tr>
<td>File Access</td>
<td>29</td>
</tr>
<tr>
<td>File Utilities</td>
<td>32</td>
</tr>
</tbody>
</table>
6. THE KERNEL

   The Event Table ........................................ 34
   The Task Table ......................................... 37
   The Event Manager ....................................... 38
       Advance ............................................... 38
       Ticket ............................................... 38
       Wait ............................................... 39
       Signals ............................................... 39
       Scheduling ......................................... 41
       Stuck and Notkt .................................... 42

   The Task Manager ....................................... 43
       A New Task ......................................... 43
       Termination ........................................ 45

   System Protection ..................................... 46

7. CONCLUSION ............................................. 48

   Improvements ........................................... 49
       Interrupts .......................................... 49
       Security ............................................ 50
       New Commands ....................................... 51

   The Future ............................................. 52

APPENDIX A ............................................... 53

REFERENCES .............................................. 62
**LIST OF ILLUSTRATIONS**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. QUAD 4 Block Diagram</td>
<td>3</td>
</tr>
<tr>
<td>2. Memory Map</td>
<td>9</td>
</tr>
</tbody>
</table>
ABSTRACT

This thesis presents a multi-tasking operating system designed for microprocessors. Task synchronization, based on naturally occurring events within the system, is discussed with an explanation of the controlling procedures.

Modularity, particularly stressed in I/O management, lead to the development of a file system which shields the user from the peculiarities of a particular device by treating all files and devices in a uniform way. This aided in the construction of a simple yet powerful I/O redirection feature.

Presently operational on an MC68000 based single-board computer, the resulting system has provided a strong incentive for future development.
CHAPTER 1

INTRODUCTION

Technological advances in high density integrated circuits have lead to the development of low cost 16-bit and 32-bit microprocessors. Along with increased word length and address range, microprocessors such as the MC68000 support a rich instruction set, have multiple registers, and vectored interrupts. These increased hardware capabilities make microprocessors suitable to applications in a "stand-alone" environment and have contributed to the recent boom in personal computers. No longer are the constraints of size, word length, and cost a major factor in microcomputer system design.

It is only natural to integrate a more powerful machine with software of comparable functionality. Unfortunately, most of the operating systems available for microcomputers today are of the single-tasking type. These systems are capable of running only one program at a time. More appropriate would be a multi-tasking system. Such a system allows the user to run as many programs concurrently as the available memory resource permits.

This thesis was motivated by a need for a portable multi-tasking system for microcomputers and the desire of
the author to understand such systems. The thesis objective was to design and implement this system using the available resources. The thesis result is QUAD 4, a real-time multitasking operating system.

Concepts

The first step in QUAD 4's development was the formulation of design goals and ideas. Portability, modularity, and ease of use were the main concern. Portability allows the system to be transported from one hardware base to another with minimal adjustments in the operating code. To achieve this, the software is written in "C". Assembly code is used only for stack and interrupt operations.

Modularity, often defined with structured programming, means dividing the system operation into separate parts each of which can be defined, coded, and debugged with little interaction between these parts. This concept is apparent in the structure of QUAD 4 which is composed of a Task manager, Event manager, Memory manager, and I/O manager (Fig. 1).

The combination of efficiency and transparency is the key to making a system easy to use. Efficiency is achieved by developing a set of simple and quick synchronization primitives. The primitives used here (advance, wait, and ticket) are based on eventcounts and sequencecounts. As defined by Reed (1979), an event is a change in the state of
command input (via terminal)

---

Console Monitor

---

command input (via terminal)

---

Command Line Interpreter

---

system calls (via user program)

---

System Call Dispatcher

---

Memory Manager

---

I/O Manager

---

Event Manager

---

Task Manager

---

Serial Device Manager

---

Disk Device Manager

---

Pipe Device Manager

---

(new device manager)

---

Fig. 1. QUAD 4 Block Diagram
one part of the system such as an interrupt caused by a serial device. An eventcount is a non-decreasing integer value which is incremented on each occurrence of a particular event.

Consider events in a bakery. First, the customer takes a ticket. If the value of the ticket is greater than the present value being served, the customer must wait. The event of importance here is the completion of a customer's service. When this event occurs, waiting customers are signaled by the advance of the service number (eventcount). The customer holding the ticket equal to the present eventcount is then served.

This method of synchronization is easily transported to tasks waiting in a queue to run or waiting to use a shared resource. Synchronization using eventcounts imposes a natural control on ordering. Its simplicity lends to the concept of a minimal kernel which is the basis for efficiency. In order to obtain a relative degree of transparency, it is necessary to conceal the internal workings of the system from the user. A fast kernel aids by minimizing the time between user input and system response.

Just as important is a unified structure of resources, as is stressed in I/O management. All I/O requests are centered around the concept of a single file entity. This implies that only one set of I/O routines are required to handle all I/O devices. Thus, the user is not
encumbered by a different set of routine calls for each device.

Included in the following text is a description of the hardware on which the system was developed, a presentation of the system, and an explanation of each of QUAD 4's managers.
CHAPTER 2

HARDWARE BASE

The QUAD 4 system was developed on Motorola's MC68000 educational computer board. This board comes equipped with two serial ports, a parallel port, 32K bytes of RAM, and a firmware package named "TUTOR". In order to accommodate QUAD 4 and user software, I expanded the RAM to 128K bytes and added a floppy disk drive. With a floppy disk as a source of mass storage, QUAD 4 could act as a separate system; not dependent on any host for operation.

TUTOR was replaced with a new console monitor (described in the next chapter). The new monitor was designed to support the expanded hardware and provide disk operations. Stored in two 8K by 8-bit EPROMs, it performs memory testing and serial port initialization on power-up.

The floppy disk interface hardware was placed on a separate board. It consists of a WD1770 floppy disk controller (FDC), an 8K by 8-bit Intel integrated RAM, and addressing logic used for direct RAM access by the FDC. This special hardware, designed by Dr. T. L. Williams and Carl Blake, frees the processor while disk read and write operations are performed.
The MC68000 microprocessor contains 8 data registers, 7 address registers, and 2 stack pointers (a user stack pointer and a supervisor stack pointer). Most of the 56 instruction types available allow byte (8-bit), word (16-bit), and long (32-bit) operations. Addressing types include register direct or indirect, immediate, absolute, program counter relative, and implied. Special addressing modes such as address register postincrement and predecrement provide for easy stack manipulation.

The processor runs under one of two privilege states; a user state or a supervisor state. Execution of instructions such as reset, stop, and those involving status register manipulations are allowed only in the supervisor state. This state is entered during interrupt or exception processing.

The MC68000 supports internal, external, and software generated exceptions. The first 1024 bytes of memory are reserved for exception vectors. The vector assignments are dedicated to machine exceptions, software traps, and 7 external interrupt auto-vectors.

One special exception is the trace exception. This is entered by setting the "T-bit" in the status register. When the T-bit is set, the next instruction is executed before the exception occurs. This is quite useful when debugging programs.
Memory Map and Management

The educational board's address decoding hardware divides the memory into 64K byte segments. The main memory or "core" resides in the first two segments. The third, fourth, and fifth segments belong to the console monitor, floppy disk controller, and serial I/O ports (Fig. 2). Since the MC68000 features a 24-bit address bus, any location can be directly addressed.

Unfortunately, the educational board did not come equipped with a memory management unit (MMU). An MMU is positioned between the processor and the system's memory. It provides memory protection, segmentation, and address translation. To overcome this deficiency, special software procedures (mentioned later in this text) had to be implemented.
<table>
<thead>
<tr>
<th>Serial I/O</th>
<th>Address 40000 to 4FFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk I/O</td>
<td>Address 30000 to 3FFFF</td>
</tr>
</tbody>
</table>
| ROM               | (Console Monitor Firmware)  
|                   | Address 20000 to 2FFFF  |
| System Location in RAM  
| Load address = 15000  |
| RAM               | (Core Memory)  
|                   | Address 0 to 1FFFF     |
| Exception Vectors  
| First lk of RAM    |

Fig. 2. Memory Map  
(all addresses in hexadecimal notation)
CHAPTER 3

THE SYSTEM

This chapter presents a survey of the QUAD 4 system. The system is divided into two sections; the console monitor and the multi-tasking system. Each section is completely independent and may be viewed as separate systems. For ease in reference, the console monitor will be called Mon and the multi-tasking system QUAD4.

Console Monitor

At system startup, the user is greeted by Mon. Mon is a single user, single terminal, console monitor with facilities for program execution and debugging. Should QUAD4 fail, it is important to have separate debugging facilities. In order to provide these, it was necessary that Mon contain its own command line interpreter, command set, I/O support, and error handling procedures.

All commands are received by polling the user terminal. When a line is entered, it is parsed into a list of null-terminated tokens. Tokens are identified as being character strings separated by any number of spaces or tabs. The command line interpreter searches Mon's list of commands for a match with the first token in the line. If a match is
found, that command is executed. Any tokens following the first token are interpreted as being arguments for that command.

Mon's command list is presented below. Arguments enclosed by '<' and '>' are optional.

boot
Load QUAD4's boot procedure from disk and execute.

d <address>
Display contents of machine registers or memory. To display memory, the starting address must be defined. If no address is present, the machine registers are displayed.

dl <address>
Download a program from a host system. This is a serial download using "S" record format. Code is loaded starting at the address specified. If the address argument is not present, code is loaded starting at the address contained in the S record address field.

dstk
Display the current stack. Depending on the current privilege state of the processor, the stack displayed may be the supervisor stack or user stack.

go <address>
Execute a program. The location of the first program instruction is either specified by the address or pointed to by the last stored value of the program counter.

mm address
Modify memory. After the memory contents are displayed, Mon waits for a modification.

mr register
Modify a register. Like modify memory, the contents of a machine register are displayed before it is modified.
rdk n s address
Read n blocks from disk 0 starting with
disk block s. The contents of the blocks
read are stored starting at the specified
address. Each disk block is 512 bytes.

set <address1> <address2>
If neither address1 nor address2 are
entered, the current breakpoint
addresses are displayed. If address1
is specified, a breakpoint is set for
that address. If both addresses are
entered, the breakpoint set at address1
is moved to address2.

sn <n>
Step n instructions. This command
used in debugging, takes advantage
of the MC68000 trace feature.

wdk n s address
Copy n blocks, from the address specified,
to disk 0 and store starting at disk
block s.

Error Handling

Two types of errors may occur; a command line error
or an execution error. A command line error may result from
either an invalid command or a mistake in syntax. An in-
valid command prompts Mon to display the list of available
commands with a brief explanation of each. If a syntax er-
ror occurs, the format for the command entered is displayed.

An execution error may be the result of a bus error,
address error, illegal instruction, privilege violation, or
zero divide. The occurrence of any of these causes a pro-
cessor generated exception. The error is reported by
displaying its type followed by the program counter contents
at the time of the error. If the error is a bus or address
error, the contents of the instruction register and address of failure are also displayed.

Booting QUAD4

The only link between Mon and QUAD4 is the boot procedure. This procedure is located in disk block 0 of the system disk. When the boot command is entered, block 0 is loaded into memory and executed.

The booting process is performed in 2 steps. First, QUAD4's configuration file is loaded from disk into memory. This file contains information on each device that is to be connected to the system. Second, QUAD4 is loaded and executed. Loading is accomplished using Mon's "rdk" command whose address is defined in the boot software.

Operating System

The last instruction in the boot procedure is a jump to sysinit. Sysinit is responsible for system initialization. This includes initializing the data structures in each of QUAD4's managers, allocating the memory occupied by the system, and connecting the I/O devices specified in the configuration file. After initialization, sysinit creates one task per user terminal. Each task is "forked" off to execute the system's command line interpreter, "Run". Run notifies the user of entrance into the multi-tasking system by the prompt '"'.

Command lines are parsed into tokens in the same fashion as in the console monitor. The first token (command token) is interpreted as being the name of an executable file and may be followed by any number of arguments. The file name may represent a resident command, nonresident command, or user program.

A resident command is located in Run's software where it is immediately executed by the "parent" task. The parent task is that task presently running the command line interpreter. Each nonresident command (user program, compiler, etc.) is executed by a separate "child" task.

Command execution may be performed in a foreground or background mode. In the foreground mode, the parent task will wait until the child has finished the execution before issuing another prompt. The background mode is entered by typing an asterisk at the beginning of the command line. In this mode the parent task creates the child and returns immediately for the next command input.

Shown below are QUAD4's resident commands. The first token is the command name. The arguments fname and fname2, may specify any file.

chd fname
   Change the current directory to fname.

delf fname
   Delete the file fname.

fidk drvn
   Format and initialize the floppy disk in drive number drvn.
ls
List the files in the current directory.

makdr fname
Make a new directory named fname.

p fname
Print the contents of fname.
P has specific knowledge of the disk format.

pr
Pr is a general purpose print command.
It prints the standard input file to the standard output file. Standard input and standard output will be discussed below.

renf fname fname2
Rename the file fname to fname2.

File I/O
A "file" is treated as a sequence of bytes, the origin of which may be a terminal, disk, pipe, or any other I/O oriented device. The particular device for which a file is associated is recognized by matching the file's name with each device's "family name".

A family name is a template for the possible file names associated with a particular class or family of devices. Each family name contains one or more "wild" cards and at least one nonalphanumeric character. Wild cards allow a variety of file names to be matched with a particular family name. The wild cards used here are asterisk and question mark. The asterisk matches any number of alphanumeric characters while the question mark matches
exactly one. Alphanumeric characters include all alphabetic and numeric characters plus underscore, period, and slash.

The present system has three I/O device drivers or device managers; the floppy disk device manager, serial device manager, and pipe device manager. Their family names are "?:*", "#?", and "|?"; respectively. Each file name shown below, matches the corresponding family name.

<table>
<thead>
<tr>
<th>File name</th>
<th>Family name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;O:foo&quot;</td>
<td>&quot;?:*&quot;</td>
</tr>
<tr>
<td>&quot;#l&quot;</td>
<td>&quot;#?&quot;</td>
</tr>
<tr>
<td>&quot;</td>
<td>a&quot;</td>
</tr>
</tbody>
</table>

(disk file name)  (serial port file name)  (pipe file name)

The disk file system is designed in a hierarchical fashion. It allows multiple directories and path names similar to the UNIX operating system (Thompson and Ritchie 1975). Unlike UNIX, each disk drive is treated as a separate file system. Thus, the root name of another disk cannot be used within a path name as an extension to the current file system. The following path name refers to the file named "foo".

"O:dir1/dir2/foo"

The root or base directory (for disk drive 0) is implied by "O:". The numeric character replacing the question mark in the family name represents the disk drive. Within the root directory is a second directory named "dir1". The last directory, "dir2", contains "foo". Here, the slash character is used to separate file names in a path and cannot be used as a character within a file name.
Each directory has within it two special file names; "." and "..". These names are entered into the directory when it is created. The file name "." refers to the current directory while ".." refers to the previous directory in the path.

The current directory is that directory in which the user is currently working. It is also the default directory. If a file name does not match any family name, it is considered (by default) to represent a file in the current directory.

The command "chd", allows the user to move from one directory to another. For instance, if a user is in directory "dir1", he could move to "dir2" by entering the following command:

```
  chd dir2
```

He could also go back to the root by substituting "dir2" with "..".

I/O Redirection

Initially, three "files" are established by the system for each new task; they are referred to as standard input, standard output, and standard error. When a command is entered at a terminal, the command line is read from the standard input "file" (usually the terminal). Command output is directed to the standard output (also usually the terminal). The input source or output destination of a
command is not restricted to the terminal and may be changed to any file. The process of changing a command's standard files is supported by Run and is referred to as I/O redirection.

There are three command line I/O redirection operators. They are <, >, and ^. The following five examples demonstrate their use.

1. `pr < foo`
   Redirect standard input from "foo" and have "foo" printed at the terminal.

2. `pr > #1`
   Redirect standard output to "foo". What is typed at the terminal is written to serial port 1.

3. `pr < foo > ../foo2`
   Copy "foo" to "foo2". The previous directory contains "foo2".

4. `pr`
   No I/O redirection! What is typed at the terminal is written back to the terminal.

5. `cmd ^ errf`
   Execute the command "cmd" and output any errors to the file "errf".

Multiple Commands

The operators ampersand (&) and vertical bar (|) are command separators. They allow more than one command to be entered on a line. Each command is executed by a separate task. Thus, if four commands are entered and separated by ampersand or vertical bar, all four will be executed simultaneously. The token immediately following either operator
is interpreted as a command and must be the name of an executable file.

Unlike the ampersand which is just a command separator, the vertical bar represents a pipe. In this implementation, a pipe is a linear buffer used as a communication channel between two cooperating tasks. It also provides synchronization between the two tasks. When a vertical bar appears between two commands, the output of the first command is directed into a pipe and the input of the second command taken from the pipe.

The following examples demonstrate the use of these operators:

1. `cmd1 > t & p foo`
2. `cmd2 < t`
3. `cmd1 | cmd2`

In example 1, the standard output of "cmd1" is directed into a temporary file. Both "cmd1" and "p" are executed simultaneously. In example 2, "cmd2" receives its input from the temporary file. In example 3, "cmd1" pipes its output directly to "cmd2". Both execute concurrently without the need for any temporary files.

A New Run

The command line interpreter may call itself. Run (the command) has two command line flags; e and p. These flags are denoted by a dash character attached as a prefix:

```
run -ep
```
The e flag is used to turn on and off command line echo. If 'e' is not present after "run", characters typed at the terminal will not be seen. The p flag turns on and off the command line prompt. If 'p' is not entered, the prompt ('~') will not be displayed. A special use for this command is presented in the next section.

In-line Files

An in-line file is treated as an ordinary file except that it is created as a part of a command. It is generated by using the standard input redirection operator ('<') followed by left and right brace characters ('{' and '}'). Contained within the braces may be any number of command lines. If more than one line is typed, the user is reminded of being in the in-line file by a right arrow following the normal prompt ("~>"'). The command line syntax of an in-line is shown below.

```
  ~ *run < {pr > foo & cmd1
  ~> chd ..
  ~> p foo2 }
```

In the above example, Run is reentered (in the background mode) to execute the in-line file. Each command line in the file is executed separately. Thus, the commands "pr" and "cmd1" are completed before the directory change which is completed before "p".
Special Characters

Certain characters have special meaning to the serial device manager. Each character commands a special service which is performed during the serial interrupt.

There are five process control characters; two for terminal display and three for command abort. By typing control-S, all serial output to a terminal is halted. A control-Q resumes the output. Commands are aborted by typing control-C (for foreground commands), control-B (for background commands), and control-D (for ending the current "run"). When control-D is entered, the parent task is killed as well as the commands for that particular run.

The delete character (DEL) and control-U are used for command line editing. DEL deletes the previous character on the line while control-U deletes the entire line.

System Calls

A system call allows a user program to enter the system for resource allocation, task synchronization, and I/O operations. System routines, available to the user, are accessed through a library of system calls linked to the user program. These calls are presented in appendix A.

A library routine enters the system through a software trap. This switches the MC68000 from the user to the supervisor privilege state. The software exception is serviced by an assembly language routine named syscall. The
library routine passes syscall an index into a table of system routine addresses. Syscall transfers the arguments passed by the user to the supervisor stack and jumps to the routine address specified by the index.

This approach to system routine access is used to separate the user program from the system. This separation is important in case of system modification. If system calls were made direct, then every user program would have to be recompiled each time the system routine is moved.
CHAPTER 4

THE MEMORY MANAGER

The necessity for memory management is not a peculiarity of multi-tasking systems. Even the simplest system must have a dedicated area for data. What makes memory management more complex, in this case, is the inability to predict the amount of memory needed and the location of available memory when a program is loaded and executed.

**Bit Map Allocation**

QUAD 4's memory management facilities are based on a bit map in which each bit represents the status of a corresponding page in memory. In this implementation, the "core memory" is mapped into 256 bits with each bit representing a page of 512 bytes. The status of a page, being either free or allocated, is determined by its corresponding bit being of value 0 or 1.

**Alloc and Free**

Two general purpose routines, alloc and free, are used for bit allocation and deallocation. They are general purpose in that they can be used with any map. Alloc is passed the address of a bit map, the size of the map, the number of bits needed in the allocation, and a pointer to
the next free area (nxtp). If the allocation succeeds, alloc returns the position in the map of the first bit allocated, else a negative one.

Allocations are made on a first fit basis starting at the position pointed to by nxtp. If nxtp is null, allocations start at bit 0. It is sometimes necessary to mark a specific area allocated. For instance, during system initialization, nxtp is set to the corresponding location in memory where the system is loaded.

Free, alloc's counterpart, is passed the same arguments as alloc plus the position of the first bit in the allocation. Free simply resets each bit in the allocated area. If nxtp is not null, it is adjusted to the position of the first free bit.

Core Allocation

The system's core is allocated using either palloc or malloc. Both routines are passed the size in bytes for a particular allocation. This size is transformed into the number bits needed in the core map and passed to alloc.

Palloc and malloc return the starting address of the allocation. They differ in that palloc keeps no allocation records while malloc manages a memory segment list for each task. A memory segment list includes the task's id and allocation history. All task allocations are handled through malloc so that if a task fails to deallocate an area the
system can do it for him. Palloc is mainly used for system allocations such as for I/O buffers.

Deallocation routines include pfree, mfree, and freeup. As their names imply, pfree and mfree simply deallocate the core allocated using palloc and malloc. All of the allocations recorded in a task's segment list are freed using freeup. Freeup is called during a task's termination to insure that he does not leave any memory allocated.

Memory Modules

User programs, nonresident commands, or any executable file read from disk has the form of a memory module. A memory module is an executable file with a module header. Each header contains the module's name, size, data area size, and offset in the module to the executable code.

The memory manager keeps an array of pointers to modules already in memory. Unless a command is recognized as being resident, it is interpreted to be the name of a memory module. The first action taken, by Run, is to create a task to "chain" to the module. The chain process involves memory module location, data area allocation, and code execution. The task checks the names of those modules in memory for a match with the command name. If a match is found a separate data area is allocated and the module's code executed. If the module is not found in memory it is loaded from disk and the memory module list updated.
Two problems may occur in attempting to load a new module; there may not be enough memory available for the module or the memory module list might be full. Included with each module pointer is a count of the module's present users. If either of the above problems occur, all unused modules are purged.

Relocation

There are certain restrictions in the generation of relocatable code when memory management hardware is not available. Data addressing or subroutine jumps cannot be absolute unless a special procedure makes address adjustments during the loading process. To avoid the overhead that would be involved, a relative addressing scheme is employed.

Before a task executes a memory module, a machine register (set aside for use as a data pointer) is initialized to the data address allocated to that task. All data is then addressed relative to that data pointer. The problem of subroutine addressing is solved by executing a jump relative to the current value of the program counter.

Many compilers and assemblers allow variables to be initialized during code generation. The initial values of these variables are stored in a data area set apart from the code. The above method does not provide a means for recognizing this data.
Although not implemented here, a process could be added that would copy a module's data into the data area allocated to the task. This would require the inclusion of data in the memory module, adding an offset to this data area in the module header, and reloading the data on each new invocation.
CHAPTER 5

THE I/O MANAGER

The I/O manager acts as an interface between I/O requests and separately defined device managers. Each device connected to the system is driven by its own device manager. This modularity enabled the development of a unified set of requests that conceal the characteristics of a particular device from the user.

A device manager is a separate module designed to drive a particular class of devices. Each manager is required to contain at least a subset of the following routines:

- `dvconnect()` /* connect the device */
- `dvopen()` /* open a file */
- `dvclose()` /* close a file */
- `dvread()` /* read from a file */
- `dwrite()` /* write to a file */
- `dvstatus()` /* report device status */
- `dvrename()` /* rename a file */
- `dvchd()` /* change current directory */
- `dvisvc()` /* perform interrupt service */

These routines support a corresponding set of service routines contained in the I/O manager. When a service is requested, the I/O manager determines which device is being addressed and diverts the request to that device manager's supporting routine.
Connect

A device manager becomes known to the I/O manager through a connect call. Connect is passed three pointers; a pointer to the device manager's family name, a pointer to the addresses of those routines supported by the manager, and a pointer to device configuration information. The first two pointers are stored in the "device manager table". They provide the information necessary for device recognition and routine addressing. The configuration pointer is passed on to dvconnect where the device manager performs its own initialization.

Device configuration is the declaration of information used for controlling the operation of a device. This information is transferred to the device manager in a data structure. The structure may include special device commands, interrupt vector addresses, physical device addresses, or any definitions that effect the response of a device to a system command or event.

File Access

A file must be opened before any access is allowed. Each open file is assigned a data structure from an array of structures called the "open file table". When a file is opened, open obtains an "empty" structure, initializes it, and records information relevant to that particular file. Shown below is the definition of this structure.
/* type definitions */
typedef unsigned Uns;  /* unsigned integer */
typedef long unsigned Luns;  /* 32 bit Uns */
typedef char *C;  /* character pointer */

/* open structure */
typedef struct {
    Uns mode;  /* mode of open */
    Luns taskid;  /* task name */
    int stflg;  /* status flag */
    Uns dvno;  /* device number */
    Uns dvfds;  /* device fds */
    Uns blksz;  /* block size */
    C badr;  /* buffer address */
    Uns blk;  /* block number */
    Uns bofs;  /* buffer offset */
    Luns filesz;  /* file size */
} OPstruct;

The index of the data structure in the open file table is called the "file descriptor". The file descriptor is returned by open to be used in place of the file name in subsequent I/O operations.

Each device manager also has its own open table. Although the type of information in each table may vary from family to family, it is used in much the same way as the open file table.

The mode of open is passed by the user. It defines how the file is to be accessed. There are three modes; read, write, and update. If a file is open for read, subsequent accesses are restricted to reading only. The same applies for write. Update allows a file to be read or written.

Two special types of an open for reading are available; an unblocked read and an open by partial match. An
unblocked read, allowed only by the serial device manager, does not force a task to wait on input when the input buffer is empty. An open by partial match allows a disk file to be opened by matching only a part of the file name.

The file status flag indicates when the file is empty. This flag is checked on each read operation. When set, an EOF (end-of-file) character is returned to the user. Thus, the user does not have to keep track of the number of bytes read to know when the file is depleted.

The I/O manager determines which device is being addressed by matching the name of the file being opened with each family name in the device manager table. If a match is found, the index in the device manager table of the matching family name is saved in dvno. On subsequent requests, the appropriate device manager and supporting routine are determined using this value.

The device file descriptor (fds), block size, and file size are obtained from the device manager's dvopen routine. The device file descriptor is the index into the device open table. This descriptor is used in the same manner as that returned by open except that it is passed by the I/O manager to the device manager.

The I/O manager supports a variety of read and write routines. Bytes can be read or written one at a time, by line, or in a user defined quantity. I/O operations for
block-oriented devices are buffered to reduce the number of device routine calls.

Single-byte transfer is supported by \texttt{readc} and \texttt{writec}. The routines \texttt{read} and \texttt{write} allow the user to define how many bytes are to be transferred. \texttt{readl} and \texttt{writel} transfer either one line or a specified number of bytes, whichever comes first. All lines are null-terminated by \texttt{readl} and may be treated as character strings.

A file descriptor passed to each routine designates what file is to be read or written. If a file is associated with a block device, such as a disk, the buffer address, block number, and offset to the next byte in that block are stored in \texttt{badr}, \texttt{blk}, and \texttt{oofs}; respectively.

When an I/O operation is complete it is the responsibility of the task that opened a file to close it. This frees the data structure by marking it empty. In some cases, such as an unexpected abort, it is not known whether a task has closed all of its files. To prevent any file from being left open, the routine \texttt{closeup} is called on exit which closes all open files with structures containing a taskid matching that of the dying task.

**File Utilities**

There are four file utilities; \texttt{seek}, \texttt{status}, \texttt{rename}, and \texttt{chd}. The function of each utility differs per device type and may not be supported by every device manager. If
not supported, a null address replaces the routine address in the device manager's routine address list.

For block-oriented devices, seek changes the position of the next read or write in a file. The position may be changed relative to the present position or to the beginning of the file. This is accomplished by determining the appropriate block number and buffer offset of the new position. Seek is also used to reposition the cursor on terminals attached to serial I/O ports.

Status returns the present file position, file size, file type, and a copy of a device's configuration information. For serial devices, the number of bytes available for reading is returned in place of the file type. The file position, file size, and configuration are returned using pointers passed as arguments. These values are optional and returned only if there associated pointers are not null.

Rename and chd are primarily supported by the disk manager. Rename is used to rename a file or delete a file. Chd is used for changing the current directory.
CHAPTER 6

THE KERNEL

The kernel is split into two managers, the task manager and event manager. QUAD 4's kernel is responsible for the system's multi-tasking capabilities. The facilities contained herein provide for task creation, termination scheduling, and event synchronization. Before discussing the implementation of these, it is important to understand the structures (the event table and task table) on which they are based.

The Event Table

As mentioned in chapter 1, an event is a change in the state of one part of the system such as the completion of a disk command or the release of a resource. The event table is a list of event structures each of which contain the history of a particular class of events. Shown below is the "C" definition of an event structure.

```c
/* 32-bit unsigned integer */
typedef long unsigned Luns;

/* event structure */
typedef struct {
    Luns evc;  /* eventcount */
    Luns tkt;  /* last ticket value */
    Luns cid;  /* last task to enter CS */
    Luns own;  /* owner */
} Ev_struct;
```
Event structures, like other resources, are allocated to a particular task. An event structure is used by tasks for synchronization.

One example of the use of these structures is as follows: The system task (a task created to perform system initialization) is allocated two event structures to synchronize access to a floppy disk. The first structure (ES1) controls access to the disk facility. The second structure (ES2) is used during the disk interrupt service to signal the completion of a disk command. Let these structures contain the following initial eventcount and ticket values: For ES1, evc1 = 1 and ttkl = 0. For ES2, evc2 = 1 and ttk2 = 1.

At some later time, a task (task1) is created to edit a file. To gain access to the disk, he must first get a ticket. In doing this, ttkl is incremented and its value returned as the ticket value. For this task to continue without waiting, evc1 must be greater than or equal to the ticket. This being the case, task1 is allowed to use the disk.

The disk read operation is performed in several command steps. Before each command is issued, the task gets a ticket from the controlling structure ES2. The eventcount and ticket values in this structure were set so that the value of the ticket obtained (2) would be greater than the eventcount (1), forcing the task to wait until the command
was complete. This allows other tasks in the system to be scheduled during each lengthy command execution.

When the disk completed the first command, an interrupt was issued which advanced the eventcount evc2. Since the ticket task1 was holding for that event was 2, he was allowed to continue.

Suppose that while task1 is waiting for the first disk command, a second task (task2) is created for the purpose of listing the current directory. Like task1, he must get a ticket to use the disk. Unlike task1, the ticket value returned is 2, making him wait until task1 is finished. This is referred to as mutual exclusion and prevents what would otherwise result, utter chaos. After task1 has finished using the disk, he increments the eventcount evcl, releasing the disk for use by task2.

There is a subtle yet important difference between the advance of evcl and that of evc2. Evc2's increment was device controlled while evcl's was task controlled. Unless the disk failed, task1 was assured of continuing. On the other hand, task2 had to depend on task1 to release the disk. If the user who was about to edit a file suddenly aborted the job, task1 may have been killed before advancing evcl thus causing task2 to wait indefinitely. This is known as a deadlock and is the reason for the third entry in the event structure cid.
When task1 was given permission to use the disk, his name (or id) was recorded in cid1. He entered what is defined as a critical section (CS); an area of code in which a deadlock can occur. If task1 were killed before advancing evcl, the system could check for the critical section entry. If the name were not found, evcl could then be advanced, releasing the disk.

The last entry in an event structure contains the id of the owner. The owner is that task allocated the event structure. When a task is terminated, all of its event structures are recognized by this entry, and deallocated.

The Task Table

The task table is an array of task structures. Each structure contains information about a particular task; its id, priority level, standard I/O descriptors, machine stack pointers, etc. In the current implementation, the task structures are linked into four lists; (1) an empty list containing structures currently unused, (2) a wait list where a task is placed when waiting on one or more events, (3) a ready list containing high priority tasks, and (4) a ready list containing low priority tasks.

During system initialization, all of the task structures are linked into the empty list and removed one at a time as new tasks are created. When an event occurs for which a task has been waiting, that task's structure is
taken from the wait list and linked to the end of its priority list where the ready tasks are dispatched in a FIFO manner. A further explanation of the scheduling process is presented in the next section describing the event manager.

The Event Manager

The event manager is the heart of the kernel. After a task has been created it is this manager that decides when the task is to wait, run, or be killed. Presented here are the synchronization primitives (advance, ticket, and wait), a signaling feature, the scheduling procedure, and the routines for handling deadlocks.

Advance

The procedure advance performs a very simple function. When passed a pointer to an event structure, it merely increments the eventcount in that structure. Although simple, this signals the system that a particular event has occurred and may result in the scheduling of one or more tasks.

Ticket

Ticket is comparable to advance in simplicity. Also passed the pointer to an event structure, it returns the incremented ticket value in that structure. This value is later used to determine when a task is ready to run.
Wait

Wait in conjunction with advance and ticket are responsible for QUAD 4's task synchronization. Passed one or more event pointers (pointer to an event structure) and corresponding ticket values, wait first checks each ticket value against the eventcount for which it is being held. If any value is less than or equal to the eventcount, the task's entry into a critical section is noted and he is allowed to continue. This avoids the overhead of scheduling on every wait. If none of the eventcounts are large enough to allow continuation, the event pointers and ticket values are stored in the task's structure, the task is linked to the wait list, and a new task is scheduled.

Signals

It is sometimes necessary to have a task perform a service he would not ordinarily perform. This is accomplished by using an event pointer and two routines; pshsvc (push service) and signal. Pshsvc is passed a pointer to an event structure and a pointer to a function which are installed in a task's service structure. When signal is called it is passed the same event pointer as passed to pshsvc. Signal determines what tasks are holding this event pointer and sets a flag in each task's structure indicating that the corresponding service function is to be executed.
During the next scheduling, each task's flag is checked. If it is set the service function is called.

One example of this is a command abort. The parent task in Run is allocated two event structures; one for foreground tasks, the other for background tasks. When a task is created to execute a command a pointer to one of these structures and a pointer to exit (an abort function) are installed in the task's service structure. The serial device manager also knows of these pointers so that when the user issues an abort character the appropriate event pointer can be passed to signal. When an abort character is typed, the task's service flag is set. During the next schedule he is forced to commit suicide.

A special use of signal is when a task is endlessly revolving in a loop that does not contain a schedule call. A single abort command would not kill this task. In this case, two consecutive abort characters cause signal to set a kill flag which instructs the serial interrupt routine to immediately call exit.

Exit is only one example of a signaled service. Any simple routine may serve, provided that it not contain any calls to wait or ticket. If a task were waiting on a particular event when forced to execute a service, a wait call during that service would result in the destruction of the previous event information and may be difficult to recover from. This not a severe restriction. If a service is
important enough that it be scheduled, it is important enough to be turned into a task.

Scheduling

The routine sched handles both scheduling and dispatching. Sched may be called directly (on a voluntary basis) or through wait. Sched's first job is to determine if any tasks have been signaled. Each task's signal flag is checked and if set, the service function for that task is executed. Sched then checks the wait list for tasks that are ready to run. If an event occurred for which a task was waiting, that task is unlinked from the wait list and linked to the end of his appropriate ready list.

As mentioned earlier there are two ready lists; one for high priority tasks and one for low priority tasks. Tasks are dispatched by first checking each ready list (high priority list first). The task at the beginning of the first non-empty list is made ready to run. This is done by performing a context-switch (or change of task). If the task was taken from the wait list, it is necessary to record the event that caused his scheduling and, his entry into a critical section. Next, a pointer to the ready list in which the task was found is advanced to the next task in the list. This allows round-robin scheduling within each priority. An assembly language routine is then called to swap the current task's frame and stack pointers with those of
the newly scheduled task. Once back from the this routine, the context-switch is complete and the newly scheduled task returns to the point where he originally called sched.

Stuck and Notkt

Two routines, stuck and notkt, are provided to aid in the systems recovery from deadlocks. A deadlock may result in either of the following cases:

1. When a task has finished using a shared resource, it is his responsibility to release the resource. If the eventcount associated with that resource is not advanced, any prior task wishing to use it will not gain entrance, thus causing a deadlock.

2. A task wishing to use a shared resource must first get a ticket. If the task does not use the resource and the ticket is discarded, he must advance the eventcount for that resource. Else, the ticket values obtained by subsequent tasks will be greater than the eventcount for that resource, preventing its use.

The distinguishing factor between case 1 and case 2 is that in case 1, the task entered a critical section and his id was recorded in the event structure for that resource. In case 2, the task did not use the resource and thus his id was never recorded.
Stuck is used to recover from case 1. This is accomplished by searching for the last task to enter the critical section. If the task is not found, it is safe to assume that the task was killed before he had a chance to advance that resource's eventcount. The eventcount may then be advanced and the resource released.

Case 2 can occur when a task is killed before entering the critical section or when a task waiting on multiple events fails to advance the eventcount of a resource he did not use. In this case, notkt (no ticket) is used for recovery. The wait list is searched for a ticket of value equal to the present eventcount for that resource. If the ticket is not found the eventcount can be advanced.

The Task Manager

At birth, each task is given its own stack, task structure, and name. At death, a clean-up process is performed which deallocates the task's accumulated resources and closes any open files. This section presents the task manager which is responsible for a task's creation and termination.

A New Task

A task is created to perform a job separate from its parent. An example being the parent task in Run creating a child to execute a user program. All tasks, except the system task which is "handcrafted", are created through fork.
Fork, called once, returns twice. The child task is recognized by returning with a different value than that of the parent.

Before the child can perform as a separate task, he must have an id, user stack, supervisor stack, and a task structure in which to store this information. A task id is a long unsigned integer. The child's new id is simply the incremented value of the last id handed out. The user and supervisor stacks are obtained by calling malloc.

In order for the child to return to the point where fork was called, it is necessary to copy the parent's stack onto the child's. Without memory management hardware, it is also necessary to adjust each frame pointer in the child's stack by an offset equal to the difference between the value of the child's supervisor stack address and his parent's. This prevents the child from switching onto his parent's stack when he returns from fork.

The child's task structure is obtained from the empty list in the task table. The first information stored in it is the child's id, stack pointers, stack addresses, priority level, and death pointer. His priority level and death pointer are passed as arguments to fork. The death pointer is a pointer to an event structure allocated by the parent. Later, when the child has completed his job, he notifies his parent by advancing the eventcount associated with the death pointer.
The child is also given his parent's standard I/O file descriptors, terminal device number, and information on the current file directory. This gives him the ability to communicate with the user.

After the child's task structure has been initialized, it is linked into the appropriate priority ready list. The parent calls sched which schedules the child to return from fork. The child is then sent out to perform his job as a separate task.

Termination

A task terminates itself by calling the procedure exit. This can be forced as in the user typing an abort character or it may occur naturally at the conclusion of the task's job. When a task dies, to insure the integrity of the system, all resources allocated by the task must be freed.

Each system manager provides a routine that releases the resources allocated to a task; the I/O manager has closeup, the memory manager has freeup, and the event manager provides evfreeup which deallocates event structures. Each of these are called by exit. Terminate, also called by exit, unlinks the task from his present list (ready or wait) and links him into the empty list.

As stated earlier, foreground and background event pointers are used to kill a task via signal. A parent task
is identified by having these event pointers in its task structure. When a parent is killed, these pointers are used to kill the surviving children, and then the associated event structures freed. If this were not done, and the event structures freed, then the surviving children could not be killed by any signal.

**System Protection**

Interrupts are the basis for asynchronous switching from one job to another. It is impossible to predict when an interrupt will occur and in what area of code the currently running task will be in. If an interrupt interferes with a task manipulating a shared variable, a "simultaneous" manipulation may occur, and must be avoided.

The following two examples describe interference caused by an interrupt. In the first example, a shared variable is used to control the entrance into an area of code that may be entered by only one task. If the variable is false, a task is allowed to enter. On entrance, the task sets the variable true in order to lock out other tasks. If an interrupt interferes with the setting of the variable, a second task may find the variable false and enter along with the first task.

A more subtle type of interference is caused by preventing a task from completing a process. Suppose a task is halted while removing a shared variable from a linked
list. If this task is killed before replacing the variable, the next task using that list may fail.

One way to prevent interference is to disable interrupts on every manipulation of a shared variable. This solution has two drawbacks. First, the system code must be thoroughly inspected in order to locate the areas where interrupts must be turned off. Second, interrupt latency is sacrificed.

A second solution is to allow interrupts with a restriction on what procedures may be executed. This method is used by QUAD 4. Both scheduling and task termination are not allowed during interrupts. This prevents the interference presented in both of the above examples and also allows the system to respond more quickly to interrupt requests.
CHAPTER 7

CONCLUSION

In the course of one year, QUAD 4 grew from a concept to an operational multi-tasking system. The original goal of designing and implementing such an executive was achieved.

A Hewlett Packard 64000 Logic Development System (HPS) was used in the initial stages of design. This system provided compilation, assembly, and debugging facilities. The HPS proved invaluable in both the hardware expansion and the creation of the console monitor.

Once the console monitor was operative and in ROM, debugging of the multi-tasking system began. The console monitor's facilities were quite efficient in locating software errors. The most useful debugging tool was the step process (command $sn$) which provided a detailed view of instruction execution.

The end product, QUAD 4, was tested using Run's commands discussed in Chapter 3. Each command was entered singly, simultaneously, and in a multiple command format. Commands executed concurrently showed little if any noticeable delay in execution. The "$pr$" command displayed the power of the modular I/O control facility. This modularity
provided a base for the I/O redirection feature. With just this one command, files could be created, copied, and messages passed between terminals.

**Improvements**

**Interrupts**

As in most newly developed systems, there is room for improvement. As stated in Chapter 6, scheduling is not allowed during interrupts. This avoids the problem of interference but requires tasks performing lengthy operations to schedule voluntarily. The initial testing of serial output made apparent the need for interrupt scheduling.

Character transmission from a serial device can be either polled or interrupt driven. The present design uses a polling method. This means that a task transmitting characters is put into a loop with its exit conditional on the transmission status of the device.

To avoid a task having dominance of the system while a file of characters is transmitted, a schedule is evoked whenever the target device is busy. But, when listing a file to a terminal using readc and writec, there is enough additional overhead in the single character transfer to allow sufficient time for a character to be transmitted between I/O requests. Thus, a context-switch never occurs.

Such is the case when executing "pr". This command was designed specifically to test this type of transfer.
When "pr" is entered with other commands simultaneously, it essentially prevents their execution until it is finished.

One solution to this problem is to provide a scheduling service that is activated by a timer interrupt. This could force scheduling at predefined intervals. Unfortunately, it would also reintroduce the possibility of interference. To prevent interference, a special flag could be set when entering an area of code that needs protection. This would signal the position of the currently running task and allow interrupt scheduling on a conditional basis.

Security

QUAD 4 does not presently furnish a login feature or file access security. To provide a login facility, a special disk file could be set aside specifically for user names. When entering Run, a procedure could prompt for the user's name and match it with those in the login file. If a match is found, the user's terminal number could be attached to his name so that future file access would be directed to that user.

Each disk file has attached a file header. This header contains the name of its owner (login name) and access information. The name in the file header is used for identification. When the user wants to open a file the file's owner name could be checked with the login name
associated with the user's terminal. If the names did not match the file would not be opened.

The file's access information could specify entrance rights. For instance, an owner could allow others to open a file by declaring it accessible for reading, writing, or both.

The file open procedure shown in appendix A, has as an argument the access type. This type is recorded in a file's header when it is created. Since only used in this case, its implementation seems rather awkward; a null must replace the access type on all other opens. It would seem more appropriate to have a default access type for file creations and a separate facility for changing it. This facility could be included in the I/O manager and added as another routine in the list of device routines (discussed in Chapter 5).

New Commands

Another improvement would be a facility for adding new commands. Presently, commands either reside in QUAD4 or are located in a user directory. A separate system directory could be used for containing a list of commands available to all users. When a command is entered, Run could locate the appropriate command code by first checking the resident commands, then the user's directory, and finally the system directory. With this feature, adding a new command would be
as easy as adding a user program. System compilation would not be required.

The Future

Presented in this thesis was the framework for future versions of the QUAD 4 system. The next version, which will test QUAD 4's portability, will be implemented on a National Semiconductor 16000 microprocessor. Although not involved in this project, the present author is anxiously awaiting its outcome.
APPENDIX A

This appendix contains QUAD 4's system calls. Each system call is presented in "C" syntax with the appropriate routine name and argument list. This appendix is not meant to be a user's guide but simply a listing of the available system calls. In some cases, the routines presented are discussed in more detail in the previous text.

System Calls

advance(evp)

arguments:
  evp - pointer to event structure

description: Advance is a synchronization primitive used to increment the eventcount associated with evp. An advance signals the occurrence of an event in the system, such as the release of a resource. An event structure and advance are discussed in Chapter 6.

alloc(map, mapsz, size, nxtp)

arguments:
  map - pointer to a bit map
  mapsz - size of the bit map (in bits)
  size - size of allocation (in bits)
  nxtp - pointer to next free area in bit map

description: Alloc is used for bit map allocations. As discussed in Chapter 4, each bit represents the status of a corresponding chunk in memory. Any resource that can be represented by a bit map may use alloc. Allocations start at the position pointed to by nxtp. If nxtp is null, allocations start at bit 0.

returns: the position of the first bit allocated or -1 if the allocation fails
chain(mnamp,a1,a2,a3,a4)

arguments:
  mnamp - pointer to a memory module name
  a1..a2 - arguments passed to memory module

description: A memory module is any executable file in memory or on disk. Chain links a task to a memory module for execution. The arguments a1 through a4 are provided for the user and mean nothing to the system.

returns: 0 or error code

chd(dnamp)

arguments:
  dnamp - pointer to directory name

description: Chd changes the current directory to the one specified by dnamp. Presently, cnd is only supported by the disk device manager.

returns: 0 or error code

close(fds)

arguments:
  fds - file descriptor of the file being closed

description: When a user is finished with a file, it should be closed. Close deallocates the resources used while the file was open.

returns: 0 or error code

config(fds,confp)

arguments:
  fds - file descriptor
  confp - pointer to device configuration structure

description: Reconfigure a device. The information contained in a device's configuration structure is passed via confp and stored by the device.

returns: 0 or error code
connect(dvnam,dvfnp,confp)

arguments:
    dvnam - pointer to device's family name
    dvfnp - pointer to list of device functions the
device manager supports
    confp - pointer to device configuration structure

description: Before a device can respond to any I/O re-
quests, it must first be connected to the system. An
explanation is contained in Chapter 5.

returns: 0 or error code

disconnect(dvnam)

arguments:
    dvnam - pointer to the device managers family name

description: Disconnect (or remove) a device from the
system.

returns: 0 or error code if device is being used

evalloc(eventcnt,tkt)

arguments:
    eventcnt - initial value of eventcount
    tkt - initial ticket value

description: Evalloc allocates an event structure. An
event structure is used for task synchronization or to
synchronize the use of a resource. The event
structure and synchronization process are discussed
in Chapter 6.

returns: pointer to an event structure or null if
none available

evfr(evp)

arguments:
    evp - pointer to event structure

description: Free an event structure.

returns: nothing
free(map, mapsz, pos, size, nxtp)

arguments:
map - pointer to a bit map
mapsz - size of the bit map (in bits)
pos - position of the first bit to be freed
size - number of bits being deallocated
nxtp - pointer to next free area

description: Free deallocates bits in a bit map. The deallocation starts at the position specified by pos. If nxtp is not null, it is adjusted to the first free bit in the map.

returns: 0 or error code if a bit is already free

malloc(size)

arguments:
size - number of bytes to be allocated

description: Malloc allocates size bytes of core memory. Each task's allocations are recorded in a segment list.

returns: address of allocation or null if no space

mfree(addr)

arguments:
addr - starting address of the area being freed

description: Free an area of memory allocated using malloc.

returns: 0 or error code if area already free
open(fnamp, mode, type)

arguments:
fnamp - pointer to a file name
mode - mode of open (read, write, update)
type - access type

description: Open a file for read, write, or update. A file must first be opened before it can be accessed. The access type is used for file creation and specifies in what mode the file may be accessed in the future.

returns: file descriptor (integer) to be used in place of the file name during subsequent I/O requests or error code

palloc(size)

arguments:
size - number of bytes to be allocated

description: Palloc allocates an area of core memory and does not keep an allocation history.

returns: address of allocation or null if no space

pfree(addr, size)

arguments:
addr - starting address of area to be freed
size - number of bytes to be freed

description: Free an area of memory allocated using palloc.

returns: error code if memory already free
read(fds, ubp, count)

arguments:
  fds - file descriptor of file to be read
  ubp - pointer to user buffer
  count - number of bytes to be read

description: Read count bytes from a file. The bytes read are transferred to the user's buffer.

returns: number of bytes read or error code

readc(fds)

arguments:
  fds - file descriptor of file to be read

description: Read one byte from a file.

returns: byte read or error code

readl(fds, ubp, count)

arguments:
  fds - file descriptor of file to be read
  ubp - pointer to user buffer
  count - number of bytes to be read

description: Read one line from a file or count bytes, which ever comes first. All lines are null terminated. The bytes read are transferred to the user's buffer.

returns: number of bytes read or error code

rename(oldp, newp)

arguments:
  oldp - pointer to old file name
  newp - pointer to new file name

description: Rename a file. The old file name is replaced by the new file name. This is primarily used for disk files, but will also clear a terminal screen if oldp points to a serial device name ("#0" is the name for port 0).

returns: 0 or error code
sched()

description: Schedule another task. This allows other
tasks to run and should be used during long uninter­
rupted processes.

returns: 0

seek(fds,newpos,smode)

arguments:
fds - file descriptor
newpos - new file position
smode - seek mode (relative or absolute)

description: Change the current file position. The new
position may be relative to the current position or
to the beginning of the file.

returns: 0 or error code

status(fds,fposp,fszp,fstp)

arguments:
fds - file descriptor
fposp - pointer for returning file position
fpszp - pointer for returning file size
fpstp - pointer for returning a copy of the
configuration structure

description: Return the status of a file. The pointer
arguments must be supplied by the user for returning
the described values. Status automatically returns
the access type of a disk file or the number of bytes
in a serial (terminal) buffer. If the serial buffer
is empty an end-of-file is returned (-1).

stdin()

description: Stdin returns a pointer to the standard
input file descriptor. The associated open file was
determined via the command line input, as discussed
in Chapter 3. A pointer is provided in case the user
wishes to change to a different input file.
stderr()

description: Stderr returns a pointer to the standard error file descriptor. As in stdin, the file descriptor may be changed.

stdout()

description: Stdout returns a pointer to the standard output file descriptor (see stdin).

ticket(evp)

arguments:
  evp - pointer to event structure

description: Ticket increments the ticket value in the specified event structure. This value is passed to wait as the event limit. When the limit is reached the task is scheduled.

returns: the incremented ticket value

wait(nev, evp0, evlim0, evpl, evliml)

arguments:
  nev - number of events to wait on
  evp0 - pointer to the associated event structure
  evlim0 - event limit
  evpl - another event to wait for
  evliml - another event limit
  (there is no limit to the number of events a task can wait for)

description: Wait suspends a task until the event for which he is waiting occurs. The task is scheduled when an eventcount reaches or exceeds the corresponding event limit (ticket). A task may wait on any number of events. An explanation of event synchronization using wait is presented in Chapter 6.

returns: a pointer to the event structure for which the task was scheduled
write(fds, ubp, count)

arguments:
  fds - file descriptor of file to be written
  ubp - pointer to user buffer
  count - number of bytes being written

description: Count bytes are taken from the user's buffer and written to the specified file.

returns: number of bytes written or error code

writec(fds, char)

arguments:
  fds - file descriptor of file being written
  char - byte being written

description: Writec writes one byte to the specified file.

returns: number of bytes written or error code

writel(fds, ubp, count)

arguments:
  fds - file descriptor of file being written
  ubp - pointer to user buffer
  count - number of bytes to be written

description: Writel transfers one line or count bytes from the user's buffer to the specified file. Lines must be terminated by a null or a new line character (control-J).

returns: returns number of bytes written or error code
REFERENCES


