CUMANA UPGRADE

OS-9 TECHNICAL REFERENCE MANUAL

Cumana Ltd., Pines Trading Estate, Broad Street, Guildford, Surrey, England, GU3 3BH. Tel: Guildford (0483) 503121. Telex: 859380.

OS-9/68000
OPERATING SYSTEM
TECHNICAL MANUAL

Revision II
Publication date: June 1987
Publication Editor: Walden Miller

Microware Systems Corporation
1900 N.W. 114th Street
Des Moines, Iowa 50322
(515)224-1929

PMN-OST68-2.111

This manual reflects 2.1 of the OS-9/68000 operating system.

PREFACE

AN OVERVIEW

OS-9/68000 is an advanced multitasking, real-time operating system for the 68000 family of microprocessors. OS-9 is well suited for a wide range of applications on 68000 computers of almost any size. It's main features are:

- Extensive management of all system resources: memory, I/O and CPU time.
- A powerful user-friendly interface.
- True multiprogramming operation.
- An expandable, device independent unified I/O system.
- Full support for modular ROMed software.

This manual is intended to provide the information necessary to install, maintain, expand and write assembly language software for OS-9 systems. It assumes that the reader is familiar with the 68000 architecture and assembly language.

High Performance

The operating system has extremely high throughput. This is primarily due to two factors:

1. The system architecture is extremely efficient, relying on re-entrant coding and careful memory management instead of disk-intensive functions.
2. The kernel, I/O system modules and device drivers are carefully optimized assembly-language system programs. Other parts of the system that are not speed critical such as the Shell and the command set are written in the C language.

Modularity

OS-9 is a highly modular operating system. It has been designed so that each module provides specific functions. The modularity of OS-9 allows individual modules to be included or deleted in the system when OS-9 is configured for a specific computer (depending on the functions that the operating system is to perform). For example, a small, ROM based control computer does not need the disk related OS-9 modules.

The operating system also has extensive support for modular software techniques. It can be easily customized or reconfigured by users. It can also be readily configured for use on almost any type of system, from small single-board computers up to large multiuser systems. OS-9 is also ROMable and has extensive support for ROMed application software.

UNIX and OS-9/6809 Compatibility

The OS-9/68000 operating system is highly compatible with the extensive base of OS-9/6809 software written in languages such as C and Microware Basic. Because OS-9's system interfaces are very similar to UNIX, most UNIX application software written in C can be compiled and run on OS-9/68000 with very minimal adaptation, if any.

TABLE OF CONTENTS

Preface

Introduction

SECTION 1 - THE KERNEL

Chapter 1 - Memory Management

Basic Functions Of The Kernel 1-1
Kernel Memory Management Functions 1-1
The Basic Module Structure 1-2
Module Requirements 1-3
Module Header Definitions 1-4
Additional Header Fields For Individual Modules 1-8
The CRC Check Value 1-11
ROMed Memory Modules 1-11
OS-9/68000 Memory Map 1-12
System Memory Allocation 1-13
Operating System Object Code 1-13
System Global Memory 1-13
System Dynamic Memory 1-13
User Memory 1-14
Memory Fragmentation 1-14

Chapter 2 - INIT & SYSGO

System Initialization From Reset 2-1
INIT: The Configuration Module 2-1
SYSGO 2-5

Chapter 3 - System Calls

System State and User State 3-1
Installing System State Routines 3-2
Kernel System Call Processing 3-3
I/O From System State 3-4
System State and Other System Calls 3-4

Chapter 4 - MPU Management & Process Execution

Overview of Multitasking 4-1
Process Memory Areas 4-2
Process Creation 4-2
Process States 4-3
Active State 4-3
Waiting State 4-5
Sleeping State 4-5
Execution Scheduling 4-5
Preemptive Task Switching 4-5

Chapter 4 - MPU Management & Process Execution (continued)
Exception And Interrupt Processing 4-6
Reset Vectors 4-8
Error Exceptions 4-8
The Trace Exception 4-9
AutoVectored Interrupts 4-9
User Traps 4-9
Vectored Interrupts 4-9

SECTION 2 - THE INPUT/OUTPUT SYSTEM

Introduction to Section 2 - The OS-9 Unified Input/Output System

Chapter 5 - File Managers
File Managers 5-1
File Manager Organization And Functions 5-1
Functions of File Manager Routines 5-2

Chapter 6 - Device Driver Modules
I/O Device Driver Modules 6-1
Basic Functional Requirements of Drivers 6-1
Driver Module Format 6-2
Interrupts and DMA 6-3
Device Descriptor Modules 6-4
Path Descriptors 6-7

Chapter 7 - Random Block File Manager
RBF Description 7-1
Disk File Physical Organization 7-1
Basic Disk Organization 7-1
Identification Sector 7-2
Allocation Map 7-3
Root Directory 7-3
Basic File Structure 7-3
Segment Allocation 7-5
Directory File Format 7-5
Raw Physical I/O On Disk-Type Devices 7-6
Record Locking 7-6
Record Locking and Unlocking 7-7
Nonsharable Files 7-7
End of File Lock 7-8
DeadLock Detection 7-8
Record Locking Details for I/O Functions 7-9
File Security 7-10

TABLE OF CONTENTS

Chapter 7 - Random Block File Manager (continued)

RBF Device Descriptor Modules 7-11
RBF Definitions Of The Path Descriptor 7-16
RBF Drivers 7-18
RBF Device Driver Storage Definitions 7-19
Device Driver Tables 7-21
RBF Device Driver Subroutines 7-22

Chapter 8 - Sequential Character File Manager

SCF Description 8-1
SCF Line Editing 8-1
SCF Device Descriptor Modules 8-1
SCF Definitions Of The Path Descriptor 8-7
SCF Drivers 8-9
SCF Device Driver Storage Definitions 8-9
SCF Device Driver Subroutines 8-13

Chapter 9 - Sequential Block File Manager

SBF Description 9-1
Tape I/O 9-1
Unbuffered I/O 9-1
Buffered I/O 9-1
End-of-tape Processing 9-2
SBF Device Descriptor Modules 9-2
SBF Definitions Of The Path Descriptor 9-4
SBF Drivers 9-5
SBF Device Driver Storage Definitions 9-5
Device Driver Tables 9-8
RBF Device Driver Subroutines 9-9

Chapter 10 - Pipe File Manager

Pipeman: The Pipe File Manager 10-1
Pipes 10-1
Named and Unnamed Pipes 10-1
Creating Pipes 10-2
Opening Pipes 10-2
Read/Readln 10-3
Write/Writln 10-3
Close 10-4
Getstat/Setstat 10-4
Pipe Directories 10-5
Pipeman Definitions of the Path Descriptor 10-6

Chapter 11 - Networking and The Network File Manager

Overview of Networking 11-1
Network Hardware Compatibility 11-1
Requirements For Networking 11-1
User Overview 11-2
Multi-station Networking 11-3
Point-to-point Networking 11-3
Multi-network Systems 11-3
The Network Utilities 11-3
Broadcast Overview 11-4
Network Security and the "_Users" Module 11-5
Building a Security Module Entry 11-6
Example Entries 11-7
Error Troubleshooting 11-8
NFM: The Network File Manager 11-10
Tracing a User Request Through a Network 11-10
The OS-9/NET Device Driver 11-13
NFM Device Descriptor Modules 11-15
NFM Path Descriptors 11-19
Node Name Data Module 11-20
NFM Device Driver Static Storage and Subroutines 11-22

Chapter 12 - The Defs Files

Using the Defs Files 12-1

SECTION 3: TRAP HANDLERS AND EVENTS

Chapter 13 - User Trap Handler Modules

Trap Handlers 13-1
Installing and Executing Trap Handlers 13-2
Two Examples: Calling a Trap Handler 13-3
An Example Trap Handler 13-5
Trace of Example Two Using the Example Trap Handler 13-7

Chapter 14 - The Math Module

Standard Function Library Module 14-1
Calling Standard Function Module Routines 14-1
Data Formats 14-2
The Math Module 14-3
The Standard Math Functions
T$Acs ArcCosine Function 14-5
T$Asn ArcSine Function 14-5
T$Atn ArcTangent Function 14-6
T$AtoD Ascii to Double-Precision Floating Point 14-6

TABLE OF CONTENTS

Chapter 14 - The Math Module (continued)

The Standard Math Functions (continued)

T$AtoF	Ascii to Single-Precision Floating Point	14-7
T$AtoL	Ascii to Long Conversion	14-7
T$AtoN	Ascii to Numeric Conversion	14-8
T$AtoU	Ascii to Unsigned Conversion	14-9
T$Cos	Cosine Function	14-9
T$DAdd	Double Precision Addition	14-10
T$DCmp	Double Precision Compare	14-10
T$DDec	Double Precision Decrement	14-11
T$DDiv	Double Precision Divide	14-11
T$DInc	Double Precision Increment	14-12
T$DInt	Round Double Precision Floating Point Number	14-12
T$DMul	Double Precision Multiplication	14-13
T$DNeg	Double Precision Negate	14-14
T$DNrm	64-bit Unsigned to Double Precision	14-15
T$DSub	Double Precision Subtraction	14-15
T$DtoA	Double Precision Floating Point to Ascii	14-16
T$DtoF	Double to Single Floating Point	14-17
T$DtoL	Double Precision to Signed Long Integer	14-17
T$DtoU	Double Precision to Unsigned Long Integer	14-18
T$DTrn	Truncate Double Precision Floating Point Number	14-18
T$Exp	Exponential Function	14-19
T$FAdd	Single Precision Addition	14-19
T$FCmp	Single Precision Compare	14-20
T$FDec	Single Precision Decrement	14-20
T$FDiv	Single Precision Division	14-21
T$FInc	Single Precision Increment	14-21
T$FInt	Round Single Precision Floating Point Number	14-22
T$FMul	Single Precision Multiplication	14-22
T$FNeg	Single Precision Negate	14-23
T$FSub	Single Precision Subtraction	14-24
T$FtoA	Single Precision Floating Point to Ascii	14-25
T$FtoD	Single to Double Floating Point	14-26
T$FtoL	Single Precision to Signed Long Integer	14-26
T$FtoU	Single Precision to Unsigned Long Integer	14-27
T$FTrn	Truncate Single Precision Floating Point Number	14-27
T$LDiv	Long (signed) Divide	14-28
T$LMod	Long (signed) Modulus	14-28
T$LMul	Long (signed) Multiplication	14-29
T$Log	Natural Logarithm Fuction	14-29
T$Log10	Comman Logarithm Function	14-30
T$LtoA	Signed Integer to Ascii Conversion	14-30
T$LtoD	Signed Integer to Double Floating Point	14-31
T$LtoF	Signed Integer to Single Floating Point	14-31

Chapter 14 - The Math Module (continued)

The Standard Math Functions (continued)

T$Power	Power Function	14-32
T$Sin	Sin Function	14-32
T$Sqrt	Square Root Function	14-33
T$Tan	Tangent Function	14-33
T$UDiv	Unsigned Divide	14-34
T$UMod	Unsigned Modulus	14-34
T$UMul	Unsigned Multiplication	14-35
T$UtoA	Unsigned Integer to Ascii Conversion	14-35
T$UtoD	Unsigned Long to Double Floating Point	14-36
T$UtoF	Unsigned Long to Single Floating Point	14-36

Chapter 15 - Events

Events 15-1

Understanding Events 15-2

Events and the F$Event System Call 15-3

SECTION 4: SYSTEM CALLS DESCRIPTIONS

Introduction to the System Call Descriptions

Chapter 16 - User State System Calls

F$AllBit	Allocate in bit map	16-1
F$Chain	Chain process to new module	16-2
F$CmpNam	Compare two names	16-4
F$CpyMem	Copy external memory	16-5
F$CRC	Generate CRC	16-6
F$DatMod	Create a data module	16-7
F$DelBit	Deallocate in bit map	16-8
F$DExec	Execute debugged program	16-9
F$DExit	Exit debugged program	16-11
F$DFork	Fork process under control of debugger	16-12
F$Exit	Terminate process	16-13
F$Fork	Start new process	16-15
F$GModDr	Get module directory copy	16-17
F$GPrDBT	Get process descriptor block table copy	16-18
F$GPrDsc	Get process descriptor copy	16-19
F$ID	Return process ID	16-20
F$Icpt	Set signal intercept	16-21
F$Julian	Get julian date	16-22
F$Link	Link to module	16-23
F$Load	Load module(s) from file	16-24
F$Mem	Set memory size	16-25

TABLE OF CONTENTS

Chapter 16 - User State System Calls (continued)

F$PErr	Print error message	16-26
F$PrsNam	Parse a path name	16-27
F$RTE	Return from interrupt exception	16-28
F$SchBit	Search bit map	16-29
F$Send	Send signal to process	16-30
F$SetCRC	Generate valid CRC in module	16-31
F$SetSys	Set/examine system global variables	16-32
F$Sleep	Put calling process to sleep	16-33
F$SPrior	Set process priority	16-34
F$SRqMem	System memory request	16-35
F$SRtMem	System memory return	16-36
F$SSpd	Suspend process	16-37
F$STime	Set current time	16-38
F$STrap	Set error Trap handler	16-39
F$SUser	Set user ID number	16-41
F$SysDbg	Call system debugger	16-42
F$Time	Set current date and time	16-43
F$TLink	Install user Trap handling module	16-45
F$UnLink	Unlink module	16-47
F$UnLoad	Unlink module by name	16-48
F$Wait	Wait for child process to terminate	16-49

Chapter 17 - I/O System Calls

I$Attach	Attach I/O device	17-1
I$ChgDir	Change working directory	17-2
I$Close	Close path	17-3
I$Create	Create new file	17-4
I$Delete	Delete file	17-6
I$Detach	Detach I/O device	17-7
I$Dup	Duplicate path	17-8
I$GetStt	Get file/device status	17-9
I$MakDir	Make directory file	17-13
I$Open	Open a path to a file or device	17-14
I$Read	Read data from a file or device	17-16
I$ReadLn	Read line of ASCII data	17-17
I$Seek	Change current position	17-18
I$SetStt	Set file/device status	17-19
I$Write	Write data to file or device	17-29
I$WritLn	Write line of ASCII data	17-30

Chapter 18 - System State System Calls

F$AllPD	Allocate process/path descriptor	18-1
F$AllPrc	Allocate process descriptor	18-2
F$AProc	Enter active process queue	18-3
F$FindPD	Find process/path descriptor	18-4
F$IOQu	Enter I/O queue	18-5
F$IRQ	Add or remove device from IRQ table	18-6
F$Move	Move data (low bound first)	18-7
F$NProc	Start next process	18-8
F$RetPD	Return process/path descriptor	18-9
F$SSvc	Service request table initialization	18-10
F$VModul	Validate module	18-12

SECTION 5: APPENDICES & INDEX

Appendix A - System Call Index

Appendix B - Examples

Appendix C - Error Codes

Index

TABLE OF CONTENTS

Figure 1: OS-9 Module Organization ii

Figure 2: Basic Memory Module Format 1-2

Figure 3: Module Header Standard Fields 1-7

Figure 4: Additional Header Fields for Individual Modules 1-10

Figure 5: Typical OS-9/68000 Memory Map 1-12

Figure 6: Additional Fields for the INIT Module 2-4

Figure 7: New Process Initial Memory Map and Register Contents 4-4

Figure 8: Beginning of a Sample File Manager Module 5-2

Figure 9: Sample Driver Module Header Format 6-3

Figure 10: Additional Standard Header Fields For Device Descriptors 6-6

Figure 11: Universal Path Descriptor Definitions 6-8

Figure 12: Identification Sector Description 7-2

Figure 13: File Descriptor Content Description 7-4

Figure 14: Initialization Table For RBF Device Descriptor Modules 7-15

Figure 15: Option Table For RBF Path Descriptor 7-17

Figure 16: RBF Static Storage Allocation 7-20

Figure 17: RBF Device Driver Table Format 7-21

Figure 18: SCF Device Descriptor Initialization Table 8-6

Figure 19: Path Descriptor Module Option Table For I/O Editing 8-8

Figure 20: Static Storage Allocation for SCF Device Drivers 8-10

Figure 21: Initialization Table for SBF Device Descriptor Modules 9-2

Figure 22: Option Table For SBF Path Descriptor Modules 9-4

Figure 23: SBF Static Storage Allocation 9-7

Figure 24: SBF Device Driver Table Format 9-8

Figure 25: Path Descriptor PD_OPT for Pipeman 10-6

Figure 26: Tracing a user request through the network 11-12

Figure 27: Message between drivers through hardware network line 11-14

Figure 28: NFM Device Descriptor Initialization Table 11-15

Figure 29: NFM Path Descriptor Option Table 11-19

Figure 30: Example Node Name Module 11-21

Figure 31: NFM Device Driver Static Storage 11-22

INTRODUCTION

OS-9 has four levels of modularity. These are described below and are shown in Figure 1.

Level 1 - The KERNEL, the CLOCK Module and the INIT Module

The KERNEL provides basic system services. These consist of I/O management, multi-tasking, memory management and linking all other system Modules.

The CLOCK Module is a software handler for the specific real-time-clock hardware. INIT is an initialization table used by the KERNEL during system startup. It specifies initial table sizes, initial system device names, etc.

Level 2 - File Managers (RBF, SCF, SBF, PIPEMAN and NFM)

File Managers perform I/O request processing for similar classes of I/O devices. The Random Block File Manager (RBF) processes all disk-type device functions. The Sequential Character File Manager (SCF) handles all non-mass storage devices. These basically operate a character at a time (i.e. printers or terminals). The Sequential Block File Manager (SBF) handles tape devices. PIPEMAN handles interprocess communication, using memory buffers for data transfer. The Network File Manager (NFM) processes data requests over the OS-9 network.

Level 3 - Device Drivers

Device Drivers handle the basic physical I/O functions for specific I/O controllers. Standard OS-9 systems are typically supplied with a disk driver, a serial port driver for terminals and serial printers, and a driver for parallel printers. Many users add customized drivers of their own design or purchase drivers from a hardware vendor.

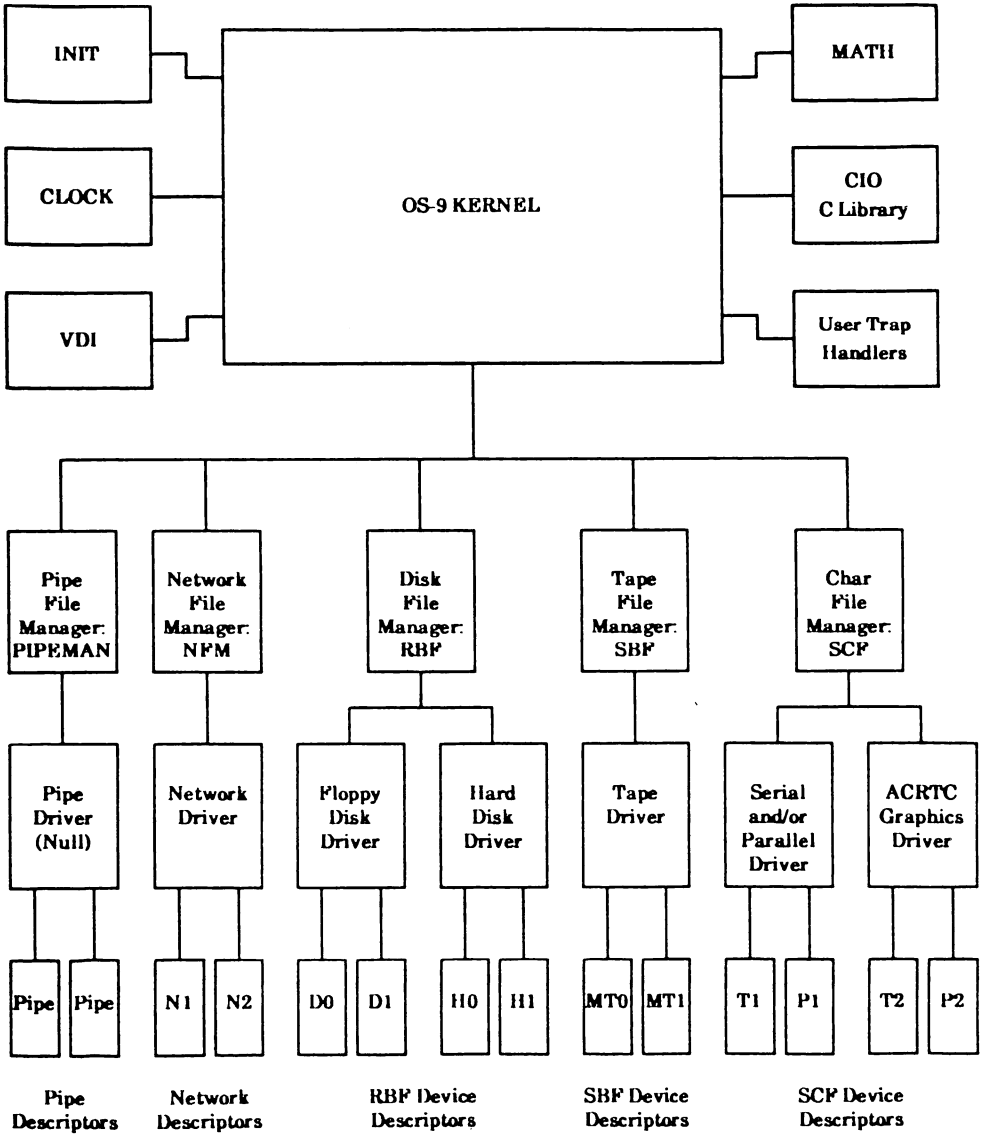
Level 4 - Device Descriptors

Device Descriptor Modules are small tables that associate specific I/O ports with their logical name, device driver and file manager. These modules also contain the physical address of the port and initialization data. By use of device descriptors, only one copy of each driver is required for each specific type of I/O device regardless of how many devices the system uses.

NOTES: One important component not shown is the Shell, which is the command interpreter. It is technically a program and not part of the operating system itself and is described fully in "Using Professional OS-9" and "Using Personal OS-9." You can see what modules make up OS-9 by using the IDENT utility on the OS9Boot file.

Even though all modules can be resident in ROM, generally only the system bootstrap module is ROMed in disk-based systems. All other modules are loaded into RAM during system startup.

Figure 1: OS-9 MODULE ORGANIZATION



SECTION 1 - THE KERNEL

- Memory Management
- INIT & SYSGO
- System Calls
- MPU Management & Process Execution

THE KERNEL - MEMORY MANAGEMENT

BASIC FUNCTIONS OF THE KERNEL

The nucleus of OS-9 is the **kernel**, which serves as the system administrator, supervisor and resource manager. It is a relatively compact module written in 68000 assembly language. It is position-independent and directly ROMable.

The kernel's main functions are:

1. Service request (system call) processing.
2. Memory management.
3. System initialization after reset.
4. MPU management (multiprogramming).
5. Input/Output management.
6. Exception and Interrupt processing.

When a system call is made, a user trap to the kernel occurs. The kernel determines what type of system call the user wants to perform.

OS-9 has two general types of system calls: calls that perform Input/Output (such as reads and writes) and calls that perform system functions, such as memory allocation and multiprogramming.

The system call functions are processed directly by the kernel. I/O calls are passed to other parts of OS-9 and are not executed by the kernel. The OS-9 system calls are discussed in detail in Chapters 16, 17 and 18.

KERNEL MEMORY MANAGEMENT FUNCTIONS

Memory management is an important operating system function. OS-9 is unique in that it manages both the physical assignment of memory to programs and the logical contents of memory by using memory modules.

Memory modules are the foundation of OS-9's modular software environment. In order for any object (a program, constant table, etc.) to be loaded into memory **it must** use the standard OS-9 memory module format. This allows OS-9 to maintain a directory which contains the name, address and other related information about each module in memory.

The operating system keeps track of modules that are in memory by the use of a module directory. When modules are loaded into memory they are added to the module directory. Each directory entry contains the address and a count of processes that are using the module. This count is called the link count.

When a process links to a module in memory, its link count is incremented by one. When a process unlinks from a module the link count is decremented by one. When a module is no longer needed (a link count of 0) its memory is deallocated and it is removed from the module directory.

The Basic Module Structure

Each module has three parts: a **module header**, a **module body** and a **CRC value** (see Figure 2):

1. The module header contains information that describes the module and its use. It is defined in assembly language by a psect directive. The header is created by the linker at link-time. The information contained in the module header includes the module's name, size, type, language, memory requirements and entry point.

2. The module body contains initialization data, program instructions, constant tables, etc.

3. The last three bytes of the module hold a CRC value (Cyclic Redundancy Check value) used to verify the module's integrity.

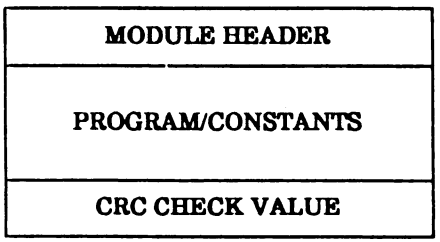


Figure 2--Basic Memory Module Format

Module Requirements

There are several different kinds of modules. Each type has a different use and function. Modules do not have to be complete programs, or even written in machine language. The main requirements are that modules do not modify themselves and that they be position-independent. This allows OS-9 to load them wherever memory space is available.

The 68000 instruction set supports a style of programming called **re-entrant code.** Modules that do not modify themselves are called re-entrant modules. This allows the exact same "copy" of a module to be shared by two or more different processes simultaneously. The processes will not affect each other, providing that each "copy" of the module has an independent memory area for its variables.

Almost all OS-9 family software is re-entrant and consequently makes most efficient use of memory. For example: Scred requires 26K bytes of memory to load. A request to run Scred is made while another user (process) is running it. OS-9 allows both processes to share the same copy, thus saving 26K of memory.

OS-9 automatically keeps track of how many processes are using each program module by its link count. It deletes a module, freeing its memory, when the module's link count becomes 0. This happens when all processes using the module have terminated.

Re-entrant code must be non-self-modifying. If one user changes a memory location, the data in that location will change for all the users of the program.

NOTE: Data modules are an exception to the "no modification" restriction. Careful coordination is required, however, for several processes to update a shared data module simultaneously.

Position-independent code means that a program does not know where it will be loaded in memory. In many operating systems, you must specify a **load address** of where the program is to be placed in memory. OS-9 determines an appropriate load address only when the program is run. OS-9 compilers and interpreters generate position-independent code automatically. In assembly language programming, however, the programmer must insure position-independence by avoiding addressing modes that refer to absolute addresses. Alternatives to absolute addressing are described in the "OS-9/68000 Assembler / Linker / Debugger User's Manual."

Module Header Definitions

Definitions of the standard set of fields in the module header are shown in Figure 3 and the following table.

NAME	UTILIZATION
M$ID	**Sync Bytes ($4AFC).** These constant bytes are used to locate modules during the startup memory search.
M$SysRev	**System revision identification.** This identifies the format of a module.
M$Size	**Size of module.** This is the overall size of the module in bytes including header and CRC.
M$Owner	**Owner ID.** This is the group/user ID of the module's "owner".
M$Name	**Offset to Module Name.** The address of the module name string relative to the start (first sync byte) of the module is located here. The name string can be located anywhere in the module and consists of a string of ASCII characters terminated by a null (zero) byte.

M$Accs	**Access Permissions.**
	These define the allowable use and access to the module by its owner or by other users. The module access permissions are divided into four sections:
	`reserved (4 bits)`
	`public (4 bits)`
	`group (4 bits)`
	`owner (4 bits)`
	The three non-reserved permission fields are defined as follows:
	`bit 3: reserved`
	`bit 2: execute permission`
	`bit 1: write permission`
	`bit 0: read permission`
	The total field is displayed as follows:
	`-----ewr-ewr-ewr`

NAME	UTILIZATION

M$Type **Module Type Code.**
Module type values are found in the "oskdefs.d" file, and describe the module type code as below:

Name	Description
	0 = Not Used (Wild Card value in system calls)
Prgm	1 = Program Module
Sbrtn	2 = Subroutine Module
Multi	3 = Multi-Module*
Data	4 = Data Module
	5-10 = Reserved*
TrapLib	11 = User Trap Library
Systm	12 = System Module (OS-9 Component)
Flmgr	13 = File Manager Module
Drivr	14 = Physical Device Driver
Devic	15 = Device Descriptor Module
	16-up = User Definable

* reserved for future use

M$Lang **Language.**
Module language codes are found in the "oskdefs.d" file. They describe whether the module is executable and which language the run-time system requires for execution (if any):

Name	Description
	0 = Unspecified Language ("Wild Card" value in system calls)
Objct	1 = 68000 machine language
ICode	2 = Basic I-code
PCode	3 = Pascal P-code
CCode	4 = C I-code*
CblCode	5 = Cobol I-code
FrtnCode	6 = Fortran I-code*
	7-15 = Reserved*
	16-255 = User Definable

* reserved for future use

NOTE: Not all combinations of module type codes and languages necessarily make sense.

NAME	UTILIZATION
M$Attr	Attributes. Bit 7 indicates that the module is re-entrant (shareable by multiple tasks). Bit 6 indicates that the module is a "ghost" module. A ghost module is retained in memory when its link count becomes zero. The module is removed from memory when its link count becomes -1 or memory is required for another use. Bit 5 indicates that the module is a "system state" module.
M$Revs	**Revision Level.** **This indicates the revision level. If two modules having the same name and type are found in the memory search or are loaded into memory, only the module with the highest Revision Level is kept. This allows easy substitution of modules for update or correction, especially ROMed modules.**
M$Edit	**Edition.** **This indicates the software release level for maintenance. Not used by OS-9. Every time a program is revised (even for a small change) this number should be increased. It is suggested that internal documentation within the source program be keyed to this system.**
M$Usage	**Comments.** Reserved for offset to module usage comments.
M$Symbol	**Symbol table offset.** Reserved for future use.
M$Parity	**Header Parity Check.** **This is the one's complement of the exclusive-OR of the previous header "words". It is used by OS-9 for a quick check of the module's integrity.**

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library, "sys.l" or "usr.l".

Offset	Usage	
$00	M$ID	Sync Bytes ($4AFC)
$02	M$SysRev	Revision ID
$04	M$Size	Module Size
$08	M$Owner	Owner ID
$0C	M$Name	Module Name Offset *
$10	M$Accs	Access Permissions
$12	M$Type	Module Type
$13	M$Lang	Module Language
$14	M$Attr	Attributes
$15	M$Revs	Revision Level
$16	M$Edit	Edit Edition
$18	M$Usage	Usage Comments Offset *
$1C	M$Symbol	Symbol Table
$20	RESERVED	
$2E	M$Parity	Header Parity Check
$30-up	Module Type Dependent	
	Module Body	
	CRC Check	

* These fields are offset to strings

Figure 3: Module Header Standard Fields

Additional Header Fields For Individual Modules

Certain types of modules have additional standard header fields following the universal offsets. These additional fields are shown in Figure 4 and the following table.

A common type of module is the "program module" (type: Prgrm, language: Objct). It is executable as an independent process by the "F$Fork" or "F$Chain" system calls. This type of module is produced by the assembler and C compilers. It is the module type of most OS-9 commands. It has six fields in addition to the universal set.

Trap handler modules are discussed in detail in Chapter 13. File Manager modules are discussed in Chapters 5. Device Drivers are discussed in Chapter 6.

NAME	UTILIZATION

(The following two fields are used by Program, Trap Handler, Device Driver, File Manager and System Module Headers.)

M$Exec **Execution Offset.**
This is the offset to the program's starting address, relative to the starting address of the module.

M$Excpt **Default user trap execution entry point.**
This is the relative address of a routine to be executed if an uninitialized user trap is called.

(The following field is used by Program, Trap Handler and Device Driver Module Headers)

M$Mem **Memory Size.**
This is the required size of the program's data area (storage for program variables).

(The following three fields are used by Program and Trap Handler Module Headers)

M$Stack **Stack Size.**
This is the minimum required size of the program's stack area.

M$IData **Initialized Data Offset.**
This is the offset to the initialization data area's starting address. This area contains values to be copied to the program's data area. All constant values declared in "vsects" are placed here by the linker. The first 4 byte value is the offset from the beginning of the data area to which the initialized data is copied. The next 4-byte value is the number of initialized data bytes to follow.

NAME	UTILIZATION

M$IRefs **Initialized References Offset.**
This is the offset to a table of values to locate pointers in the data area. Initialized variables in the program's data area may contain values that are pointers to absolute addresses. Code pointers must be adjusted by adding the absolute starting address of the object code area. The data pointers must be adjusted by adding the absolute starting address of the data area. This effective address calculation is done automatically by the F$Fork system call at execution time using tables created in the module which contain the following information:

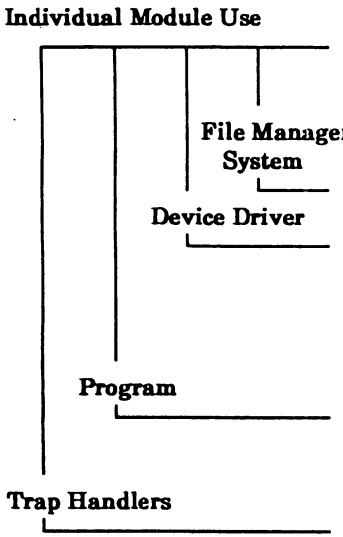
M$IRefs **Initialized References Offset. (continued)**
The first word of each table is the most significant (MS) word of the off-set to the pointer. The second word is a count of the number of least significant (LS) word offsets to be adjusted. The adjustment is made by combining the MS word with each LS word entry. This offset locates the pointer in the data area. The pointer is adjusted by adding in the absolute starting address of the **object code** or the **data area** (for code pointers or data pointers respectively). It is possible after exhausting this first count that another MS word and count are given. This continues until a MS word of zero and a count of zero is found.

(The following two fields are used by Trap Handler Module Headers)

M$Init **Initialization Execution Offset.**

M$Term **Termination Execution Offset.**

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l".



Offset		Usage
$30	M$Exec	Execution Offset
$34	M$Excpt	Default User Trap Execution Entry Point
$38	M$Mem	Memory Size
$3C	M$Stack	Stack Size
$40	M$IData	Initialized Data Offset
$44	M$IRefs	Initialized References Offset
$48	M$Init	Initialized Execution Offset
$4C	M$Term	Termination Execution Offset

Figure 4: Additional Header Fields for Individual Modules

The CRC Check Value

At the end of all modules is a CRC or Cyclical Redundancy Check value. The CRC is an error checking method used frequently in data communications and storage systems. It is used to check the validity of the entire module. It is also a vital part of the ROM memory module search technique. It provides a very high degree of confidence that programs in memory are intact before execution. It serves as an almost foolproof backup for the error detection systems of disk drives, memory systems, etc.

In OS-9, a 24-bit CRC value is computed over the entire module starting at the first byte of the module header and ending at the byte just before the CRC itself. OS-9 family compilers and assemblers automatically generate the module header and CRC values. If required, a user program can use the F$CRC system call to compute a CRC value over any specified data-bytes. For a full description of how F$CRC computes a module's CRC, refer to the F$CRC system call description.

OS-9 will not recognize a module with an incorrect CRC value. For this reason, you must update the CRC value of any module "patched" or otherwise modified in any way, or the module can not be loaded from disk or found in ROM. The OS-9 utility, FIXMOD, can be used to update the CRC's of patched modules.

ROMed Memory Modules

When OS-9 starts after a system reset, the kernel searches for modules in ROM. It detects them by looking for the module header sync code ($4AFC). When this byte pattern is detected, the header parity is checked to verify a correct header. If this test succeeds, the module size is obtained from the header and a 24 bit CRC is computed over the entire module. If the computed CRC is valid, the module is entered into the module directory. The chances of detecting a "false module" are virtually nil.

OS-9 links to any of its component modules that were found during the search. All ROMed modules present in the system at startup are automatically included in the system module directory. This allows partially or completely ROM-based systems to be created. ROMs containing non-system modules that are found are also included. This allows user-supplied software to be located during the start-up process and entered into the module directory.

OS-9/68000 MEMORY MAP

OS-9 uses a software memory management system where all memory is contained within a single memory map. Therefore, all user tasks share a common address space.

A map of a typical OS-9/68000 memory space is shown in Figure 5. The various sections shown for ROM, RAM, I/O, etc., are not required to be at specific addresses (except where noted). We recommend that each section be kept in contiguous reserved blocks arranged in an order that facilitates future expansion. It is always advantageous for RAM to be physically contiguous as much as possible.

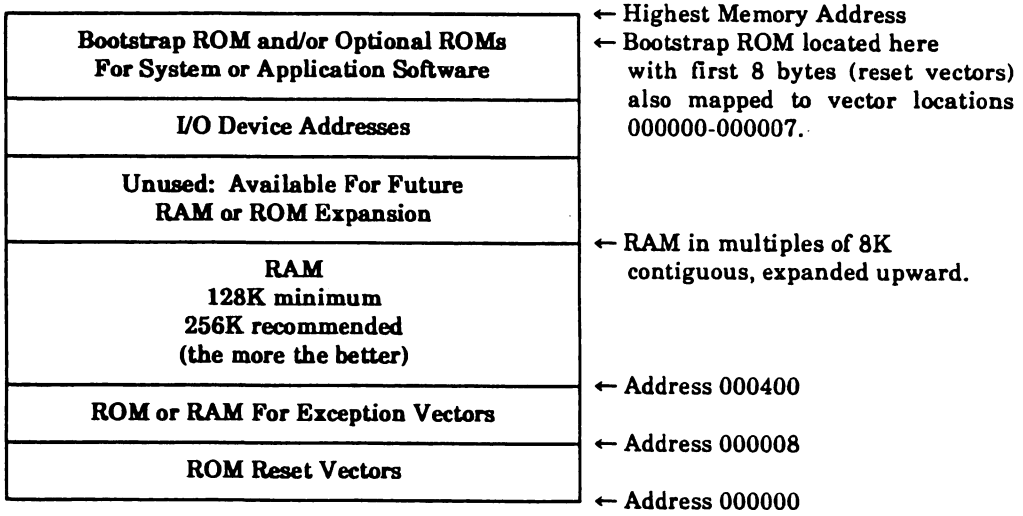


Figure 5: Typical OS-9/68000 Memory Map

SYSTEM MEMORY ALLOCATION

During the OS-9 start-up sequence, blocks of RAM and ROM are found by an automatic search function in the Boot Rom. Some RAM is reserved by OS-9 for its own data structures. ROM blocks are searched for valid OS-9 ROM modules.

The amount of memory required by OS-9 is variable. Actual requirements depend on the system configuration and the number of active tasks and open files. The following sections describe approximate amounts of memory used by various parts of OS-9.

Operating System Object Code

A complete set of typical operating system component modules (kernel, I/O managers, device drivers, etc.) occupies about 24K to 32K bytes of memory. These modules are normally bootstrap loaded into RAM on disk-based systems. OS-9 **does not** dynamically load overlays or swap system code so no additional RAM is required for system code.

Alternatively, OS-9 can also be placed in ROM for non-disk systems. The typical operating system object code size for ROM-based, non-disk systems is about 20K - 24K bytes.

System Global Memory

OS-9 uses an 8K section of RAM memory for internal use. This memory area is usually located at the lowest RAM memory addressed. It contains an exception jump table and system global variables. Variables in this area are symbolically defined in the "sys.l" library using name prefixes of "D_".

WARNING: Despite the temptation, user programs should **never** directly access these variables. System calls are provided to allow user programs to read the information in this area.

System Dynamic Memory

OS-9 maintains dynamic-sized data structures (such as I/O buffers, path descriptors, process descriptors, etc.) which are allocated from the general RAM area when needed. Pointers to the addresses of these data structures are kept in the System Global Memory area. On a typical small system, the RAM memory used will be approximately 16K. Exact sizes of all the system's data structure elements can be found by studying a listing of the source files that make up the "sys.l" library file.

User Memory

All unused RAM memory is assigned to a free memory pool. Memory space is removed and returned to the pool as it is allocated or deallocated for various purposes. OS-9 automatically assigns memory from the free memory pool whenever any of the following occur:

1. When modules are loaded into RAM.
2. When new processes are created.
3. When processes request additional RAM.
4. When OS-9 needs more I/O buffers or its internal data structures must be expanded.

Storage for user program object code modules and data space is dynamically allocated from and deallocated to the free memory pool. User object code modules are also automatically shared if two or more tasks execute the same object program. User object code application programs can also be stored in ROM memory.

The total memory needed for user memory largely depends on the application software to be run. It is suggested that a system minimum of at least 128K plus an additional 64K per user be available. Alternatively, a small ROM-based control system might only need 32K of memory.

MEMORY FRAGMENTATION

Once a program is loaded it must remain at the address at which it was originally loaded. Even though position independent programs can be initially placed at any address where free memory is available, program modules can not be relocated dynamically afterwards. This characteristic can lead to a sometimes troublesome phenomenon called "memory fragmentation".

When programs are loaded, they are assigned the first sufficiently large block of memory at the highest address possible in the address space. If a number of program modules are loaded, and subsequently one or more non-contiguous modules are "unlinked", several fragments of free memory space will exist. The total free memory space may be quite large. But because it is scattered, not enough space will exist in a single block to load a particular program module.

End of Chapter 1

SYSTEM INITIALIZATION FROM RESET

After a hardware reset, the kernel (located in ROM or loaded from disk, depending on the system involved) is executed by the "bootstrap" ROM. The kernel then initializes the system. This includes locating ROM modules and running the system startup task (usually SYSGO).

INIT: THE CONFIGURATION MODULE

INIT is a module that contains system startup parameters. It is a table used during startup to specify initial table sizes and system device names. It must be in memory when the kernel is executed and usually resides in the OS9Boot file.

INIT is a non-executable module. It has the module type: "Systm" (code $0C). OS-9 uses INIT to configure itself during startup. It is always available to determine system limits. The module begins with a standard module header (Figure 3 in Chapter 1) and the additional fields shown in Figure 6 and the following table.

NOTE: See Appendix B for an example program listing of the INIT module. Offset names are defined in the relocatable library "sys.l".

NAME	DESCRIPTION
M$PollSz	**Number of entries in the IRQ polling table.** This is the number of entries in the IRQ polling table. One entry is required for each interrupt generating device control register.
M$DevCnt	**Device table size.** This is the number of entries in the system device table. One entry is required for each device in the system.
M$Procs	**Initial process table size.** This indicates the initial number of active processes allowed in the system. If this table becomes full, it will automatically expand as needed.
M$Paths	**Initial path table size.** This is the initial number of open paths in the system. If this table becomes full, it will automatically expand as needed.
M$SParam	**Offset to parameter string for startup module** This is the offset to the parameter string (if any) to be passed to the first executable module.

NAME	DESCRIPTION
M$SysGo	**First executable module name offset.** This is the offset to the name string of the first executable module; usually SYSGO or Shell.
M$SysDev	**Default directory name offset.** This is the offset to the initial default directory name string (usually /D0 or /H0). The sysem initially does a "chd" to this device and expects to find a directory named "CMDS" and a file named "startup" on it. If the system does not use disks this offset must be zero.
M$Consol	**Initial I/O pathlist name offset.** This is the offset to the initial I/O pathlist string (usually /TERM). This pathlist is opened as the standard path for the initial startup module. It is generally used to set up the initial I/O paths to and from a terminal. This offset may contain zero if it not used.
M$Extens	**Customization module name offset.** This is the offset to the name string of a customization module (if any). A customization module is intended to be used to complement or change the existing standard system calls used by OS-9. This module is searched for at startup, and if found executed in system state. Typically it is found in the bootfile. The default name string to be searched for is "OS9P2".
M$Clock	**Clock module name offset.** This is the offset to the clock module name string.
M$Slice	**Timeslice.** The number of clock ticks per timeslice.
M$Instal	**Offset to installation name.** This is the offset to the installation name string.
M$CPUTyp	**Cpu Type.** Cpu type: 68000, 68008, 68010, 68020.

NAME	DESCRIPTION

M$OS9Lvl **Level, Version and Edition.**
This four byte field is divided into three parts:

```
level:  1 byte     version:  2 bytes     edition:  1 byte
```

For example, level 2, version 2.0, edition 0 would be 2200.

M$OS9Rev **Revision offset.**
This is the offset to the OS-9 level/revision string.

M$SysPri **Priority.**
This is the system priority that the first module (usually SYSGO or Shell) is executed at. This is generally the base priority that all processes start at.

M$MinPty **Minimum Priority.**
This is the initial system minimum executable priority. For a complete discussion on Minimum Priority, see Chapter 4 on execution scheduling and Chapter 17 (F$SetSys).

M$MaxAge **Maximum Age.**
This is the initial system maximum natural age. For a complete discussion on Maximum Age, see Chapter 4 on execution scheduling and Chapter 17 (F$SetSys).

M$Events **Number of Entries in the Events Table**
This is the initial number of entries allowed in the events table. If the table becomes full, it will automatically expand as needed. See Chapter 16 on Events for discussion of event usage.

M$Compat **Revision Compatability**
This byte is used for revision compatibility. The following bits are currently defined:

Bit 0 set to save all registers for IRQ routines
Bit 1 set to ignore "ghost" bit in module headers
Bit 2 set to prevent the kernal from using "stop" instructions

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

Offset		Usage
$30	Reserved	
$34	M$Pollsz	# IRQ Polling Table Entries
$36	M$DevCnt	Device Table Size
$38	M$Procs	Initial Process Table Size
$3A	M$Paths	Initial Path Table Size
$3C	M$SParam	Parameter String For Startup Module (Usually Sysgo)
$3E	M$SysGo	First Executable Module *
$40	M$SysDev	Default Directory *
$42	M$Consol	Initial Standard I/O Path *
$44	M$Extens	Customization Module Name *
$46	M$Clock	Clock Module Name *

Offset		Usage
$48	M$Slice	Ticks per Time Slice
$4A	Reserved	
$50	M$Instal	Installation Name *
$52	M$CPUTyp	CPU Type
$56	M$OS9Lvl	Operating System Level
$5A	M$OS9Rev	Revision Name *
$5C	M$SysPri	Initial System Priority
$5E	M$MinPty	Minimum Priority
$60	M$MaxAge	Maximum Age
$62	Reserved	
$66	M$Events	Initial Event Table Size
$68	M$Compat	Revision Compatibility
$69	Reserved	(28 bytes)

* These fields are offsets to name strings.
The strings themselves follow the 28-byte reserved section.

Figure 6: Additional Fields For The INIT Module

SYSGO

SYSGO is the first user process started after the system startup sequence. Its standard I/O will be on the system console device. It usually performs the following functions:

1. SYSGO does additional high level system initialization. For example, SYSGO will call the Shell to process the startup shell procedure file.

2. SYSGO starts the console Shell (or other program).

3. SYSGO remains in a wait state during system operations. This acts as insurance against all processes terminating, leaving the system halted. For example, if the console terminal process terminates, SYSGO generally restarts it.

The standard SYSGO module for disk systems can not be used on non-disk systems. However, it is easy to customize SYSGO if necessary.

SYSGO may be eliminated entirely by specifying "shell" as the initial startup module and specifying a parameter string similar to:

```
startup; ex tsmon /term
```

See Appendix B for an example source listing of the SYSGO module.

NOTES

SYSTEM STATE AND USER STATE

Before discussing OS-9's system calls, two terms must be understood: User state and System state. These are the two distinct OS-9 environments in which object code can be executed:

User State: User state is the normal program environment in which processes are executed. Generally, user state processes never deal directly with the specific hardware configuration of the OS-9 machine.

System State: The OS-9 system calls and interrupt service routines run in system state. On 68000-family processors, this is synonymous with "supervisor" state. System state routines often deal with physical hardware present on a system.

Functions that execute in system state have several distinct advantages over those running in user state:

1. A system state routine has access to the entire capabilities of the processor. For example, on memory protected systems, a system state routine may access any memory in the system. It may mask interrupts, alter OS-9 internal data structures or take direct control of hardware interrupt vectors if necessary.

2. There is a group of OS-9 system calls that are accessible only from system state.

3. System state routines are never timesliced. Once a process has entered system state, no other process will execute until the system state process has finished. Only if the system state process goes to sleep (i.e. F$Sleep waiting for I/O), will other processes execute. The only processing that may pre-empt a system state routine is interrupt servicing.

The characteristics of system state make it the only way to provide certain types of programming functions. For example, it is almost impossible to provide direct I/O to a physical device from user state. For this reason, users occasionally wonder why all programs are not run in system state. There are several reasons why this should not be done:

1. In a multi-user environment, it is important to ensure that each user receives a fair share of the CPU time. This is the basic function of time-slicing.

2. Memory protection prevents user state routines from directly accessing I/O devices, but it also prevents user state routines from accidentally damaging data structures they do not own.

3. A user state process may be aborted. If a system state routine loses control, the entire system almost always crashes.

4. It is far more difficult and dangerous to debug system state routines than user state routines. The user state debugger can be used to find most user state problems. Some system state problems are much more difficult to find.

5. User programs almost never have to be concerned with physical hardware; they are essentially isolated from it. This makes user state programs more portable and easily written.

Installing System State Routines

With direct access to all system hardware, any system state routine has the power to completely take over the entire machine. This much power can be dangerous. In a system state routine, it is often a challenge to keep from crashing or hanging up the system. For this reason, the methods of creating routines that operate in system state have been limited.

On OS-9, there are four obvious ways of providing system state routines:

1. Install an "OS9P2" module in the system bootstrap file (or in ROM). During cold start, the OS-9 kernel will link to this module, and if found, call its execution entry point. The most likely thing for such a module to do is install new system call codes. There is one drawback to this method: the OS9P2 module must be in memory when the system is bootstrapped.

2. Use the I/O system as an entry into system state. File managers and device drivers are always executed in system state. In fact, the most obvious reason to write system state routines is to provide support for new hardware devices. It is possible to write a dummy device driver and use the I$GetStt or I$SetStt routines to provide a gateway to the driver. Once the driver is called, it can of course install new OS-9 system calls if it likes.

3. Write a trap handler module that executes in system state. For routines of limited use that are to be dynamically loaded and unlinked, this is perhaps the most convenient method. In many cases, it is practical to debug most of the trap handler routines in user state and then convert the trap module to system state. To make a trap handler execute in system state, it is necessary to set the "supervisor" state bit in the module attribute byte and create the module as super user. When the user trap is executed, it will be in system state.

4. A program will execute in system state if the "supervisor" state bit in the module's attribute/revision word is set and if the module is owned by the super user. In some rare instances, this can be useful.

IMPORTANT REMINDER: System state routines are not timesliced, and therefore should be written as short and fast as possible.

KERNEL SYSTEM CALL PROCESSING

All OS-9 service requests (system calls) are processed through the kernel. Service requests are used to communicate between OS-9 and assembly language programs for such things as allocating memory or creating processes. In addition to I/O and memory management functions, there are other service request functions that include interprocess control and timekeeping.

The system wide relocatable library files, "sys.l" and "usr.l" define symbolic names for all service requests. The files are linked with hand-written assembly language or compiler-generated code. The OS-9 Assembler has a built-in macro to generate system calls:

```
OS9     I$Read
```

This is recognized and assembled to produce the same code as:

```
TRAP    #0
dc.w    I$Read
```

In addition, the C Compiler standard library includes standard functions to access nearly all user mode OS-9 system calls from C programs.

Parameters for system calls are usually passed and returned in registers. System calls are divided into three categories:

1. **User State System Calls:** These functions perform memory management, multiprogramming and other functions for user programs. These are mainly processed by the kernel. The symbolic names for this category begin with "F$". For example, the system call to link a module is called F$Link.

2. **I/O System Calls:** These requests perform various I/O functions and are processed in the File Manager and Device Driver for a particular device. The symbolic names for this category begin with "I$". For example, the "read" service request is called "I$Read".

3. **System State System Calls:** These requests are special system calls that can only be used by system software in system state. They usually operate on internal OS-9 data structures. They are system calls instead of subroutines in order to preserve the OS-9's modularity. These calls can not be accessed by user state programs. They are documented in this manual for programmers who may use them when writing system modules such as device drivers. The symbolic names for these system calls begin with "F$".

I/O From System State

All of the I/O system calls may be used from a system state routine. They have one slight difference than when executed in user state. The difference is that all path numbers used in system state are "system" path numbers. Each process descriptor has a path number that is used to convert process local path numbers into system path numbers. The system itself has a global path number table used to convert system path numbers into actual addresses of path descriptors. System state I/O system calls must be made using system path numbers.

For example, the OS-9 F$PErr system call prints an error message on the caller's standard error path. To do this, it may not simply perform output on path number two. Instead it must examine the caller's process descriptor and extract the system path number from the third entry (0, 1, 2, ...) in the caller's path table.

When a user state process exits with I/O paths open, they are automatically closed by the F$Exit routine. This is possible because OS-9 has kept track of the open paths in the process' path table. In system state, the I$Open and I$Create system calls return a system path number. It is not recorded in the process path table or anywhere else by OS-9. This makes it the responsibility of the system state routine that opens any I/O paths to ensure that they eventually become closed. This is true even if the underlying process is abnormally terminated.

System State And Other System Calls

In general, system state routines may use any of the ordinary (user state) system calls. Care must be taken, however, to avoid making system calls at inappropriate times. For example, an interrupt service routine should avoid I/O calls, timed sleep requests and other calls that can be particularly time consuming (F$CRC).

Any memory requested in system state is not recorded in the process descriptor memory list. This makes it the responsibility of the requester to ensure that the memory is returned to the system before the process terminates.

WARNING: System state routines should avoid the F$TLink and F$Icpt system calls. Certain portions of the C library may be inappropriate for use in system state.

End of Chapter 3

THE KERNEL - MPU MANAGEMENT & PROCESS EXECUTION

OVERVIEW OF MULTITASKING

OS-9 is a multitasking operating system which allows several independent programs called **processes** or **tasks** to be executed simultaneously. All OS-9 programs are run as processes. Each process can have access to any system resource by issuing appropriate service requests to OS-9.

CPU time is a finite resource that must be allocated efficiently to maximize the computer's throughput. Characteristically, many programs spend much unproductive time waiting for various events to occur (such as an input/output operation).

A good example is an interactive program which communicates with a person at a terminal. While the program waits for a line of characters to be typed or displayed, it (typically) can not do any useful processing and may waste valuable CPU time.

An efficient multiprogramming operating system such as OS-9 automatically assigns CPU time to only those programs that can effectively use the time. To accomplish this, OS-9 uses a technique called **timeslicing.**

Timeslicing allows processes to share CPU time with all other active processes. It is implemented by using both hardware and software functions.

The system's CPU is interrupted by a real time clock at a regular rate of (usually) 100 times each second. This basic time interval is called a "tick". Therefore, the interval between ticks is usually 10 milliseconds.

At any occurrence of a tick, OS-9 can suspend execution of one program and begin execution of another. The starting and stopping of programs is done in a manner that does not affect the program's execution.

Processes that are **active** (not waiting for some event) are run for a specific system assigned period called a **time slice.** How often a process receives a time slice depends on a process' priority value relative to the priority of all other active processes. Many OS-9 service requests are available to create, terminate and control processes.

This technique is called timeslicing because each second of CPU time is sliced up to be shared among several processes. Timeslicing happens so rapidly that to a human observer, all processes appear to execute continuously (unless the computer becomes overloaded with processing).

If overloading does occur, a noticeable delay in response to terminal input may result or "batch" programs may take much longer to run than they ordinarily do.

Process Memory Areas

All processes are divided into two logically separate memory areas: one area for code and one for data. This dichotomy provides OS-9's modular software capabilities.

A program must be in the form of an executable memory module (described in detail in Chapter 1) in order to be run. In this form, it is called the **primary module.** It may link to and execute code in other modules. It is position-independent and ROMable, and the memory it occupies is considered to be "read-only".

The process' **data area** is a separate memory space where all of the program's variables are kept. The top part of this area is used for the program's stack. The actual memory addresses that are assigned to the data area are not known at the time the program is written. A base address is kept in a register (usually a6 by convention) to access the data area. This area can be read or written to.

If a program uses variables that require initialization, the initializing values must be copied from the read-only program area to the data area where the variables actually reside. The OS-9 linker builds appropriate initialization tables which are used by OS-9 to initialize the variables.

Process Creation

New processes are created by the F$Fork ("fork") system call. The most important parameter passed in the fork system call is the name of the "primary module" that the new process is to initially execute. The creation process is outlined as follows:

1. **Locate or Load the Program.** OS-9 first tries to find the module in memory. If it can not be found, OS-9 loads into memory a mass-storage file using the requested module name as a file name.

2. **Allocate and Initialize a Process Descriptor.** Once the primary module has been located, a data structure called a **process descriptor** is assigned to the new process. The process descriptor is a table that contains information about the process, its state, memory allocation, priority, I/O paths, etc. The process descriptor is automatically initialized and maintained by OS-9. The process need not be concerned about the descriptor's existence or what it contains.

3. **Allocate the Stack and Data Areas.** The primary module's module header contains a data and stack size. OS-9 allocates a **contiguous** memory area of the required size from the free memory space.

4. **Initialize the Process.** The new process' registers are set up to the proper addresses in the data area and object code module (see Figure 7). If the program uses initialized variables and/or pointers, they are copied from the object code area to the proper addresses in the data area.

If any of these steps can not be performed, creation of the new process is aborted, and the process that originated the "fork" is informed of the error. Otherwise, the new process is added to the active process queue for execution scheduling.

The new process is also assigned a unique number called a **Process ID** which is used as itsidentifier. Other processes can communicate with it by referring to its ID in various system calls. The process also has an associated **Group ID** and **User ID** which are used to identify all processes and files belonging to a particular user and group of users. The IDs are inherited from the parent process.

Processes terminate when they execute an "F$Exit" system service request or when they receive fatal signals or errors. The process termination closes any open paths, deallocates its memory, and unlinks its primary module.

PROCESS STATES

At any instant, a process can be in one of three states:

ACTIVE The process is active and ready for execution.

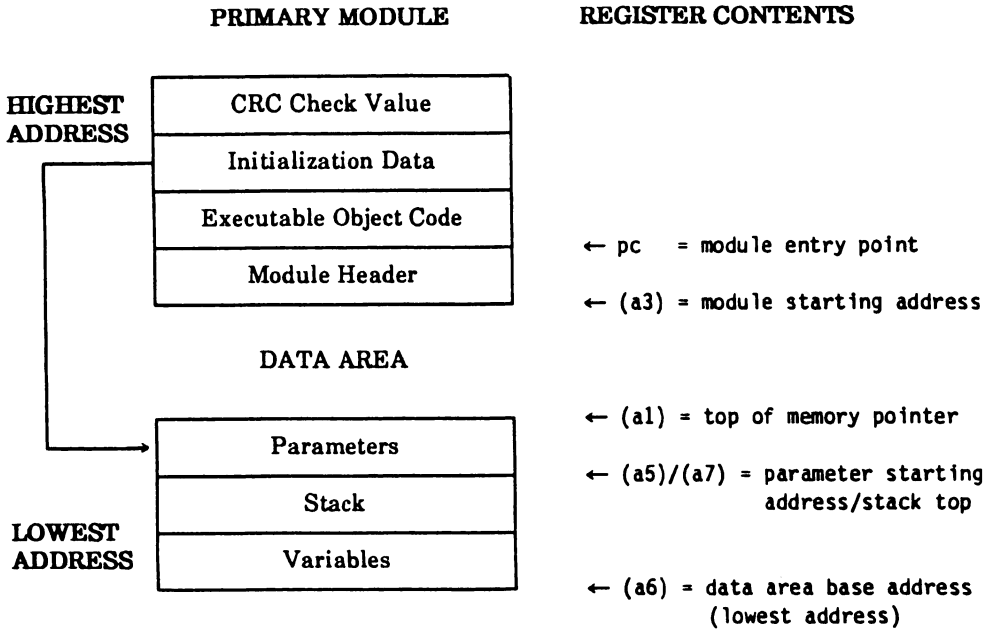
WAITING The process is inactive until a child process terminates or a signal is received.

SLEEPING The process is inactive for a specific period of time or until a signal is received.

There is a queue for each process state. Each queue is a linked list of process descriptors corresponding to all processes with the same process state. State changes are performed by moving a process descriptor from its current queue to another queue.

The Active State

The active state includes all executable processes. These processes are given time slices for execution according to their relative priority with respect to all other active processes. The scheduler uses a method that involves using an age comparison with each active process in the queue. It gives all active processes some CPU time, even if they have a very low relative priority (see the following section on Execution Scheduling).



Registers passed to the new process:

```
sr   = 0000                                (a0) = undefined
pc   = module entry point                  (a1) = top of memory pointer
d0.w = process ID                          (a2) = undefined
d1.l = group/user ID                       (a3) = primary module pointer
d2.w = priority                            (a4) = undefined
d3.w = # of I/O paths inherited            (a5) = parameter pointer
d4.l = undefined                           (a6) = static storage (data area) base
d5.l = parameter size                             pointer
d6.l = total initial memory allocation     (a7) = stack pointer (same as a5)
d7.l = undefined
```

NOTE: (a6) is always biased by $8000 to allow object programs to access 64K of data using indexed addressing. Usually this bias can be ignored because the Linker automatically adjusts for it.

Figure 7: New Process Initial Memory Map And Register Contents

The Waiting State

The wait state is entered when a process executes a F$Wait system service request. The process remains inactive until any of its descendant processes terminates or until it receives a signal.

The Sleeping State

Sleep state is entered when a process executes an F$Sleep service request. The F$Sleep request specifies a time interval for which the process is to remain inactive. Processes often do this to avoid wasting CPU time while waiting for some external event (such as the completion of I/O). Zero ticks specifies an infinite period of time. The process remains asleep until the specified time has elapsed or until a signal is received.

EXECUTION SCHEDULING

The kernel is responsible for allocation of CPU time to active processes. OS-9 uses a scheduling algorithm that ensures all processes get some execution time.

All active processes are members of the **active process queue**, which is kept sorted by **process age**. Process age is a count of how many process switches have occurred since the process entered the queue, plus the process' initial priority.

When a process is moved to the active process queue, its age is initialized by setting it equal to the process' assigned priority. Processes having relatively higher priority are placed in the queue with an artificially higher age. Whenever a new process is placed in the active queue the ages of all other processes are incremented. Ages are never incremented beyond $FFFF.

Upon conclusion of the currently executing process' time slice, the scheduler selects the process having the highest age to be executed next. Because the queue is kept sorted by age, the oldest process will be at the head of the queue.

Pre-emptive Task Switching

During critical real-time applications, fast interrupt response time is sometimes necessary. OS-9 provides this by pre-empting the currently executing process when a process with a higher priority becomes active. The lower priority process loses the remainder of its time slice. It is then re-inserted into the active queue.

Task switching is affected by two system global variables: D_MinPty and D_MaxAge. Both variables are accessible by users with a group ID of zero (super users) through the F$SetSys system call.

D_MinPty defines a minimum priority below which processes are neither aged nor considered candidates for execution. D_MinPty is usually set to zero on most systems. When it is set to some priority level, all processes running below that level are stopped completely while the critical task (or tasks) runs to completion.

This is potentially dangerous because if the minimum system priority is set above the priority of all running tasks, the system will be completely shut down and can only be recovered by a reset. It is crucial to restore D_MinPty to a normal level when the critical task finishes.

D_MaxAge defines a maximum age that processes are not allowed to mature above. Usually, this variable is set to zero, but when it is activated, D_MaxAge essentially divides tasks into two classes: low and high priority. Low priority tasks stop aging at the MaxAge cutoff. The high priority task (or tasks) will receive the entire available CPU time. High priority tasks are never aged. All low priority tasks will be run only when the high priority task(s) are inactive.

The exception to these rules is that any process that is performing a system call will not be pre-empted until the call is finished, unless it voluntarily gives up its timeslice. This exception is made because these processes may be executing critical routines that affect shared system resources and therefore could be blocking other unrelated processes.

EXCEPTION AND INTERRUPT PROCESSING

One of OS-9/68000's nicer features is its extensive support of the 68000's advanced exception/interrupt system. Routines to handle a particular exception can be installed using various OS-9 system calls for the different types of exceptions.

Vector Number	Related OS-9 Call	Assignment
0	none	reset initial SSP
1	none	reset initial PC
2	F$STrap	bus error
3	F$STrap	address error *
4	F$STrap	illegal instruction *
5	F$STrap	zero divide *

* See section heading "Error Exceptions"

Vector Number	Related OS-9 Call	Assignment
6	F$STrap	CHK instruction
7	F$STrap	TRAPV instruction
8	F$STrap	privilege violation *
9	F$DFork	trace
10	F$STrap	line 1010 emulator *
11	F$STrap	line 1111 emulator *
12-13	none	(reserved) *
14	none	(reserved) format error *
15	none	uninitialized interrupt *
16-23	none	(reserved) *
24	none	spurious interrupt *
25-31	F$IRQ	level 1-7 interrupt autovectors
32	F$OS9	user TRAP #0 instruction (OS-9 call)
33-47	F$TLink	user TRAP #1 - #15 instruction vectors
48	F$STrap	FPCP Branch or set on unordered condition *
49	F$STrap	FPCP Inexact result *
50	F$STrap	FPCP Divide by zero *
51	F$STrap	FPCP Underflow error *

* see section heading "Error Exception"

Vector Number	Related OS-9 Call	Assignment
52	F$STrap	FPCP Operand Error *
53	F$STrap	FPCP Overflow error *
54	F$STrap	FPCP NAN signaled *
55-63	none	reserved
64-255	F$IRQ	vectored interrupts

* see section heading "Error Exception"

Reset Vectors: vectors 0, 1

The reset initial SSP vector contains the address loaded into the system's stack pointer at startup. There must be 4k of RAM below and at least 4k of RAM above this address for system global storage. Each time any exception occurs, OS-9 uses this vector to find the base address of system global data.

The reset initial PC is the coldstart entry point to OS-9. Its only use after startup is to reset after a catastrophic failure.

WARNING: User programs should not use or alter either of these vectors.

Error Exceptions: vectors 2-8, 10-24, 48-63

These exceptions are usually considered fatal program errors and cause a user program to be unconditionally terminated. If F$DFork created the process, the process resources will remain intact and control will return to the parent debugger to allow a post-mortem examination.

The F$STrap system call may be used to install a user subroutine to catch the errors in this group that are considered non-fatal.

When an error exception occurs, the routine is executed in user state, with a pointer to the normal data space used by the process and all user registers stacked. The exception handler must decide whether and where to continue execution.

If any of these exceptions occur in system state, it usually means a system call has been passed bad data and an error is returned. In some cases, system data structures can be damaged by passing nonsense parameters to system calls.

NOTE: Vectors 48-53 occur only on 68020 systems with a floating point 68881 math coprocessor.

The Trace Exception: vector 9

The trace exception occurs when the status register trace bit is set. This allows the MPU to single step instructions. OS-9 provides the F$DFork, F$DExec and F$DExit system calls to control program tracing.

AutoVectored Interrupts: vectors 25-31

These exceptions provide interrupt polling for I/O devices that do not generate vectored interrupts. Internally, they are handled exactly like vectored interrupts.

WARNING: Level 7 interrupts should not normally be used, because they are non-maskable and can interrupt the system at dangerous times. Level 7 interrupts may be used for "software refresh" of dynamic RAMs or similar functions. The IRQ service routine for this vector may not use any OS-9 system calls or system data structures.

User Traps: vectors 32-47

The system reserves user trap zero (vector 32) for standard OS-9 system service requests. The remaining 15 user traps provide a method to link to common library routines at execution time.

Library routines are similar to program object code modules, and are allocated their own static storage when installed by the F$TLink service request. The execution entry point is executed whenever the user trap is called. In addition, trap handlers have initialization and termination entry points, which are executed when linked and at process termination.

Vectored Interrupts: vectors 64-255

The 192 vectored interrupts provide a minimum amount of system overhead in calling a device driver module to handle an interrupt. Interrupt service routines are executed in system state without any associated current process. The device driver must provide an error entry point for the system to execute if any error exceptions occur during interrupt processing. The F$IRQ system call is used to install a handler in the system's interrupt tables. Multiple devices may be used on the same vector if necessary.

End of Chapter 4

NOTES

SECTION 2 - THE INPUT/OUTPUT SYSTEM

- File Managers
- Device Driver Modules
- Random Block File Manager
- Sequential Character File Manager
- Sequential Block File Manager
- Pipe File Manager
- Network File Manager
- The Defs Files

INTRODUCTION TO SECTION 2

THE OS-9 UNIFIED INPUT/OUTPUT SYSTEM

OS-9 features a versatile unified, hardware-independent I/O system. The I/O system is modular. It can be easily expanded or customized by the user.

The kernel performs some I/O processing (such as allocating data structures), and then calls an appropriate file manager module, which may in turn call a device driver module. The file manager, device driver and device descriptor modules are standard memory modules that can be installed and removed dynamically while the system is running.

The kernel provides the first level of service for I/O system calls by routing data between processes and the appropriate file managers and device drivers. It maintains two important internal OS-9 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 device name given (or implied) in the pathlist. The module to be linked to is the device's descriptor, which contains the names of the device driver and file manager for the device. The information in the device descriptor is saved by the kernel so subsequent system calls can be routed to these modules.

FILE MANAGERS

The function of a file manager is to process the raw data stream to or from device drivers for a class of similar devices. The file manager makes a device driver conform to the OS-9 standard I/O and file structure by removing as many unique device operational characteristics as possible from I/O operations. File managers are also responsible for mass storage allocation and directory processing if applicable to the class of devices they service.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They may also monitor and process the data stream. For example, they may add line feed characters after carriage return characters.

The file managers are re-entrant. One file manager may be used for an entire class of devices having similar operational characteristics. OS-9 systems can have any number of File Manager modules.

File managers should have the system state bit set in the attribute byte of module header. OS-9 does not make use of this currently, however future revisions will erquire File managers to be system state modules.

The four file managers which are included in typical systems are:

1. **RBF (Random Block File Manager):** This manager operates random-access, block-structured devices such as disk systems.

2. **SCF (Sequential Character File Manager):** This manager is used with single-character-oriented devices such as CRT or hardcopy terminals, printers and modems.

3. **PIPEMAN (Pipe File Manager):** This manager supports interprocess communication through memory buffers called "pipes".

4. **SBF (Sequential Block File Manager):** This manager is used with sequential block-structured devices such as tape systems.

FILE MANAGER ORGANIZATION AND FUNCTIONS

A file manager is a collection of major subroutines accessed through an offset table. The table contains the starting address of each subroutine relative to the beginning of the table. The location of the table is specified by the execution entry point offset in the module header. A sample listing of the beginning of a file manager module is listed in Figure 8.

When the individual file manager routines are called, standard parameters are passed in the following registers:

(a1) Pointer to Path Descriptor
(a4) Pointer to current Process Descriptor
(a5) Pointer to User's Register Stack; User registers pass/receive parameters as shown in the system call description section
(a6) Pointer to system Global Data area

```
*  Sample File Manager
*  Module Header declaration
       Type_Lang equ (FlMgr<<8)+Objct
       Attr_Revs equ (ReEnt+Supstat<<8)+0

       psect    FileMgr,Type_Lang,Attr_Revs,Edition,0,Entry_pt

*  Entry Offset Table
Entry_pt  dc.w     Create-Entry_pt
          dc.w     Open-Entry_pt
          dc.w     MakDir-Entry_pt
          dc.w     ChgDir-Entry_pt
          dc.w     Delete-Entry_pt
          dc.w     Seek-Entry_pt
          dc.w     Read-Entry_pt
          dc.w     Write-Entry_pt
          dc.w     ReadLn-Entry_pt
          dc.w     WriteLn-Entry_pt
          dc.w     GetStat-Entry_pt
          dc.w     SetStat-Entry_pt
          dc.w     Close-Entry_pt
*  Individual Routines Start Here
```

Figure 8: Beginning Of A Sample File Manager Module

FUNCTIONS OF FILE MANAGER ROUTINES

Create, Open

Open and Create are responsible for opening or creating a file on a particular device. This typically involves allocating any buffers required, initializing path descriptor variables, and parsing the path name. If the file manager controls multi-file devices (RBF), directory searching is performed to find or create the specified file.

Makdir

Makdir creates a directory file on multi-file devices. Makdir is neither preceded by a Create nor followed by a Close. File managers that are incapable of supporting directories, return with the carry bit set and an appropriate error code in (d1.w).

ChgDir

On multi-file devices, ChgDir searches for a file which must be a directory file. If the directory is found, the address of the directory is saved in the caller's process descriptor at P$DIO.

Specifically, the RBF File Manager saves the address of the directory's file descriptor sector. Open/Create begins searching in this directory when the caller's pathlist does not begin with a "/" character.

File managers that do not support directories return with the carry bit set and an appropriate error code in (d1.w).

Delete

Multi-file device managers usually do a directory search that is similar to Open and, once found, remove the file name from the directory. Any media that was in use by the file is returned to unused status.

File managers that do not support multi-file devices simply return an error.

Seek

File managers that support random access devices use Seek to position file pointers of the already open path to the byte specified. Typically, this is a logical movement and does not affect the physical device. No error is produced at the time of the seek, if the position is beyond the current "end of file".

File managers that do not support random access usually do nothing, but do not return an error.

Read

Read is responsible for returning the number of bytes requested to the user's data buffer. It returns an EOF error, if there is no data available. Read must be capable of copying pure binary data, and generally performs no editing on the data. Usually, the file manager will call the device driver to actually read the data into a buffer. It then copies data from the buffer into the user's data area. This method helps keep file managers device independent.

Write

The Write request, like Read, must be capable of recording pure binary data without alteration. Usually, the routines for read and write are almost identical with the exception that Write uses the device driver's output routine instead of the input routine. Writing past the end of file on a device expands the file with new data.

RBF and similar random access devices that use fixed-length records (sectors) must often pre-read a sector before writing it unless the entire sector is being written.

ReadLn

ReadLn differs from Read in two respects. First, ReadLn is expected to terminate when the first end-of-line character (carriage return) is encountered. Second, ReadLn performs any input editing that is appropriate for the device.

Specifically, the SCF File Manager performs editing that involves handling backspace, line deletion, echo, etc.

WriteLn

WriteLn is the counterpart of ReadLn. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing also is performed. After a carriage return, for example, SCF usually outputs a line feed character and nulls (if appropriate).

Getstat, Setstat

The Getstat (Get Status) and Setstat (Set Status) system calls are wild card calls designed to provide a method of accessing features of a device (or file manager) that are not generally device independent.

The file manager may perform some specific function such as setting the size of a file to a given value. Status calls that are unknown by the file manager are passed on to the driver to provide a further means of device independance. For example, a SetStat call to format a disk track may behave differently on different types od disk controllers.

Close

Close is responsible for ensuring that any output to a device is completed (writing out the last buffer if necessary), and releasing any buffer space allocated when the path was opened. It does not execute the device driver's terminate routine, but may do specific end-of-file processing if necessary, such as writing end-of-file records on tapes.

NOTES

I/O DEVICE DRIVER MODULES

I/O Driver modules perform basic low-level physical input/output functions. For example, a disk driver module's basic functions are to read or write a physical sector. The driver is not concerned about files, directories, etc., which are handled at a higher level by the OS-9 file manager. Device driver modules are re-entrant so one copy of the module can simultaneously support multiple devices that use identical I/O controller hardware.

This section describes the function and general design of OS-9 device driver modules to aid programmers in modifying existing drivers or writing new ones. In order to present this information in an understandable manner, only basic drivers for character-oriented (SCF-type) and disk-oriented (RBF-type) devices are discussed. It is suggested that you study this section in conjunction with a source listing of a sample device driver.

Basic Functional Requirements of Drivers

A driver module is actually a package of seven subroutines that are called by a file manager in system state. Their functions are:

1. Initialize the device controller hardware and related driver variables as required.
2. Read a standard physical unit (a character or sector, depending on the device type).
3. Write a standard physical unit (a character or sector, depending on the device type).
4. Return a specified device status.
5. Set a specified device status.
6. De-initialize the device. It is assumed that the device will not be used again unless re-initialized.
7. Process an error exception generated during driver execution.

When written properly, a single physical driver module can handle multiple identical hardware interfaces. The specific information for each physical interface (port address, initialization constants, etc.) is given in a small **device descriptor** module.

Driver Module Format

All drivers must confirm to the standard OS-9 memory module format. The module type code is "Drivr". Drivers should have the system state bit set in the attribute bute of the module header. OS-9 does not make use of this currently, however future revisions will require all device drivers to be system state modules. A sample assembly language header is shown in Figure 9.

The execution offset in the module header (M$Exec) gives the address of an offset table, which specifies the starting address of each of the seven driver subroutines.

The static storage size (M$Mem) specifies the amount of local storage required by the driver. This is the sum of the storage required by the file manager (V_xxx variables) plus any variables and tables declared in the driver.

The driver subroutines are called by the associated file manager through the offset table. The driver routines are always executed in system state. Regardless of the device type, the standard parameters listed below are passed to the driver in registers. Other parameters that depend on the device type and subroutine called may also be passed. These are described in individual chapters concerning file managers.

INITIALIZE and TERMINATE:

(a1) the address of the device descriptor module.
(a2) the address of the driver's static variable storage.
(a4) the address of the process descriptor requesting the I/O function.
(a6) the address of the system global variable storage area.

READ, WRITE, GETSTAT and SETSTAT:

(a1) the address of the path descriptor.
(a2) the address of the driver's static variable storage.
(a4) the address of the process descriptor requesting the I/O function.
(a5) a pointer to the calling process' register stack
(a6) the address of the system global variable storage area.

ERROR: This entry point should be defined as the offset to exception handling code or zero if no handler is available is available. This entry point is currently not used by the kernal. However, in future revisions this will be accessed. 6-2

Each subroutine is terminated by an RTS instruction. Error status is returned using the CCR carry bit with an error code returned in register d1.w.

```
* Module Header

            Type_Lang equ (Drivr<<8)+Objct
            Attr_Revs equ (ReEnt<<8)+0

            psect  Acia,Typ_Lang,Attr_Rev,Edition,0,AciaEnt

* Entry Point Offset Table
   AciaEnt dc.w   Init          Initialization routine offset
           dc.w   Read          Read routine offset
           dc.w   Write         Write routine offset
           dc.w   GetStat       Get dev status routine offset
           dc.w   SetStat       Set dev status routine offset
           dc.w   TrmNat        Terminate dev routine offset
           dc.w   Error         Error handler routine offset
```

Figure 9: Sample Driver Module Header Format

Interrupts and DMA

Because OS-9 is a multitasking operating system, optimum system performance will be obtained when all I/O devices are set up for interrupt-driven operation.

For character-oriented devices, the controller should be set up to generate an interrupt upon the receipt of an incoming character and at the completion of transmission of an out-going character. Both the input data and the output data should be buffered in the driver.

In the case of RBF-type device, the controller should be set up to generate an interrupt upon the completion of a sector read or a sector write operation. It is not necessary for the driver to buffer data because the driver is passed the address of a complete buffer. DMA sector transfers improve data transfer speed significantly.

Usually, the INIT routine adds the relevant device interrupt service routine to the OS-9 interrupt polling system using the F$IRQ system call. The controller interrupts are enabled and disabled by the READ and WRITE routines as may be required.

The following interrupt priority levels are recommended:

```
Real-Time Clock         Level 6
Terminal/Printer Ports  Level 4
Disk Controllers        Level 3
I/O Processors          Level 2
```

DEVICE DESCRIPTOR MODULES

Device descriptor modules are small, non-executable modules that provide information that associates a specific I/O device with its logical name, hardware controller address(es), device driver name, file manager name and initialization parameters.

Device drivers and file managers both operate on general classes of devices, not specific I/O ports. The device descriptor modules tailor their functions to a specific I/O device. One device descriptor module must exist for each I/O device in the system. However, one device may also have several device descriptors with different initialization constants.

The name of the module is used as the logical device name by the system and user (i.e. it is the device name given in pathlists). Its format consists of a standard module header that has a type code of "device descriptor" (Devic). The remaining header fields are shown in Figure 10, and described in the following table.

NOTE: These fields are standard for all Device Descriptor Modules. They are followed by a device specific initialization table (see Chapters 7 through 11 for the File Manager specific Device Descriptor Module tables).

NAME	UTILIZATION
M$Port	**Port address.** This is the absolute physical address of the hardware controller.
M$Vector	**Trap Vector.** 25-31 for an autovectored interrupt. 64-255 for a vectored interrupt.
M$IRQLvl	**IRQ Hardware Interrupt Level.**
M$Prior	**IRQ Polling Priority.** Smaller numbers are polled first if more than one device is on the same vector. A priority of zero indicates that the device requires exclusive use of the vector.
M$Mode	**Device Mode Capabilities** This byte is used to validity check a caller's access mode byte in I$Create or I$Open calls. If a bit is set, the device is capable of performing the corresponding function. The ISize_bit is usually set, because is it usu-ally handled by the file manager or ignored. If the Share_bit (Single User bit) is set here, the device will be non-sharable. This is useful for printers.

NAME	UTILIZATION
M$FMgr	**File Manager Name offset.** This is the offset to the name string of the File Manager module to be used.
M$PDev	**Device Driver Name offset.** This is the offset to the name string of the Device Driver Module to be used.
M$DevCon	**Device Configuration.** Reserved.
M$Opt	**Table Size.** This contains the size of the initialization table.
M$DTyp	**Initialization table.** This table is Device specific. M$DTyp must be the first byte of the option table. It is a code to indicate what type of device this is. M$DTyp values usually correspond to a particular file manager.

The initialization table is copied into the "option section" of the path descriptor when a path to the device is opened. The values in this table may be used to define the operating parameters that are accessible by the I$GetStat and I$SetStat system calls. For example, a terminal's initialization parameters define which control characters are used for backspace, delete, etc. The maximum size of the initialization table is 128 bytes.

You may wish to add additional devices to your system. If an identical device controller already exists, all you need to do is add the new hardware and another device descriptor. Device descriptors can be in ROM, in the boot file, or loaded into RAM from mass-storage files while the system is running.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: "sys.l" or "usr.l."

Offset		Usage
$30	M$Port	Port Address
$34	M$Vector	Trap Vector Number
$35	M$IRQLvl	IRQ Interrupt Level
$36	M$Prior	IRQ Polling Priority
$37	M$Mode	Device Mode Capabilities
$38	M$FMgr	File Manager Name Offset
$3A	M$PDev	Device Driver Name Offset
$3C	M$DevCon	Device Configuration Offset
$3E	Reserved	
$46	M$Opt	Initialization Table Size
$48	M$DTyp	Device Type

Figure 10: Additional Standard Header Fields For Device Descriptors

PATH DESCRIPTORS

Every open path is represented by a data structure called a path descriptor ("PD"). It contains information required by file managers and device drivers to perform I/O functions. Path descriptors are dynamically allocated and deallocated as paths are opened and closed.

PDs have three sections: the first section is defined universally for all file managers and device drivers, as shown in the Figure 11:

The section called "PD_FST" is reserved for and defined by each type of file manager for file pointers, permanent variables, etc.

The 128 byte section called "PD_OPT" is used as an "option" area for dynamically-alterable operating parameters for the file or device. These variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module, and can be examined or altered later by user programs by means of the GETSTAT and SETSTAT system calls.

Current definitions of the option area for specific-type devices are given in the description of the particular file manager. These are included in the "sys.l" or "usr.l" library file, and are linked into programs that need them.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable libraries, "sys.l", or "usr.l."

Offset		Usage
$00	PD_PD	Path Number
$02	PD_MOD	Access Mode (R W E S D)
$03	PD_CNT	# of Paths using this PD
$04	PD_DEV	Address of Related Device Table Entry
$08	PD_CPR	Requester's Process ID
$0A	PD_RGS	Address of Caller's MPU Register Stack
$0E	PD_BUF	Address of Data Buffer
$12	PD_USER	Group/User ID of Original Path Owner
$16	PD_FST	File Manager Working Storage
$80	PD_OPT	Option Table

Figure 11: Universal Path Descriptor Definitions

End of Chapter 6

RANDOM BLOCK FILE MANAGER

RBF DESCRIPTION

The Random Block File Manager (RBF) is a re-entrant subroutine package for I/O service requests to random-access devices. Specifically, RBF is a file manager module that supports random-access, block-oriented mass storage devices (disk systems, bubble memory systems, and high-performance tape systems). RBF can handle any number or type of such systems simultaneously. It is responsible for maintaining the logical and physical file structures.

RBF is designed to support a wide range of devices having different performance and storage capacities. Consequently, it is highly parameter driven.

The physical parameters it uses are stored on the media itself. On disk systems, this information is written on the first few sectors of track number zero. The device drivers also use this information, particularly the physical parameters stored on sector 0. These parameters are written by the "FORMAT" program that initializes and tests the media.

DISK FILE PHYSICAL ORGANIZATION

The RBF file manager supports a tree structured file system. The physical disk organization was designed to be efficient in use of disk space, highly resistant to accidental damage, and to allow fast file access. The system also has the advantage of relative simplicity.

Basic Disk Organization

The OS-9 standard sector size is 256-byte sectors. If a disk system is used that can not directly support 256-byte sectors, the driver module must divide or combine sectors as required to simulate 256-byte size.

Most disks are physically addressed by track number, surface number and sector number. In order to eliminate hardware dependencies, OS-9 uses a **logical sector number** (LSN) to identify each sector without regard to track and surface numbering.

It is the responsibility of the disk driver module or the disk controller to map logical sector numbers to track/surface/sector addresses. OS-9's file system uses LSNs from 0 to n-1 ("n" = the total number of sectors on the drive). All sector addresses discussed in this section refer to LSNs.

The FORMAT utility is used to initialize the file system on blank or recycled media by creating the track/surface/sector structure. In the process, the media is tested for bad sectors which are automatically excluded from the file system.

Every OS-9 disk has the following basic structure:

The Identification Sector is located in Logical Sector Zero (LSN 0). It contains a description of the physical and logical format of the storage volume (disk media).

The Disk allocation Map usually begins in Logical Sector One. This indicates which disk sectors are free and available for use in new or expanded files.

The Root Directory of the volume begins immediately after the allocation map.

Identification Sector

LSN zero always contains the identification sector (see Figure 12). It describes the physical format of the disk and the location of the other parts of the file system (allocation map and root directory). It also contains the volume name, date and time of creation, etc. If the disk is a bootable system disk it will also have the starting LSN and size of the "OS9Boot" file.

Addr	Size	Name	Description
$00	3	DD_TOT	Total number of sectors on media
$03	1	DD_TKS	Track size in sectors
$04	2	DD_MAP	Number of bytes in allocation map
$06	2	DD_BIT	Number of sectors/bit (cluster size)
$08	3	DD_DIR	LSN of root directory file descriptor
$0B	2	DD_OWN	Owner ID
$0D	1	DD_ATT	Attributes
$0E	2	DD_DSK	Disk ID
$10	1	DD_FMT	Disk Format; density/sides Bit 0: 0 = single side 1 = double side Bit 1: 0 = single density 1 = double density Bit 2: 0 = single track (48 TPI) 1 = double track (96 TPI)
$11	2	DD_SPT	Sectors/track (two byte value DD_TKS)
$13	2	DD_RES	Reserved for future use
$15	3	DD_BT	System bootstrap LSN
$18	2	DD_BSZ	Size of system bootstrap
$1A	5	DD_DAT	Creation date
$1F	32	DD_NAM	Volume name
$3F	32	DD_OPT	Path descriptor options

Figure 12: Identification Sector Description

Allocation Map

The allocation map shows which sectors have been allocated to files and which are free for future use.

Each bit in the allocation map represents a sector on the disk or a cluster of sectors. If a bit is set, the sector is considered to be in use, defective or non-existent. The allocation map usually starts at LSN one and uses a variable number of sectors according to how many bits are needed for the map. DD_MAP (see Figure 12) specifies the actual number of bytes used in the map.

Each bit in the map corresponds to a cluster of sectors on the disk. The number of sectors per cluster is specified by the DD_Bit variable and is always an integral power of two.

Multiple sector allocation maps allow the number of sectors per cluster to be as small as possible for high volume media. The Format utility sets the size of the allocation map depending on the size and number of sectors per cluster. The number of sectors per cluster can be selected on the command line when the Format utility is invoked.

Root Directory

This file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as "/D1"). Usually, it immediately follows the allocation map. The location of the root directory FD is specified in DD_DIR (see Figure 12).

Basic File Structure

OS-9 uses a multiple-contiguous-segment type of file structure. **Segments** are physically contiguous sectors used to store the file's data. If all the data can not be stored in a single segment (because a file is expanded after creation, or a sufficient number of contiguous free sectors are not available), additional segments are allocated to the file.

The OS-9 segmentation method was designed to keep a file's data sectors in as close physical proximity as possible in order to minimize disk head movement. Frequently, files (especially small files) will have only one segment. This will result in the fastest possible access time. Therefore it is good practice to initialize the size of a file to its expected maximum size during or immediately after its creation. This will allow OS-9 to optimize its storage allocation.

All files have a sector called a **file descriptor sector** (or FD). An FD contains a list of the data segments; their starting LSNs and sizes. It is also where information such as the file attributes, owner and time of last access is stored. This sector is used only by the system and is not directly accessible by the user. The table in Figure 13 describes the contents of a file descriptor.

NOTE: The term "offset" refers to the location of a field, relative to the starting address of the File Descriptor. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

Offset	Size	Name	Description
$00	1	FD_ATT	File Attributes: D S PE PW PR E W R
$01	2	FD_OWN	Owner's User ID
$03	5	FD_DAT	Date Last Modified: Y M D H M
$08	1	FD_LNK	Link Count
$09	4	FD_SIZ	File Size (number of bytes)
$0D	3	FD_CREAT	Date Created: Y M D
$10	240	FD_SEG	Segment List: see below

Figure 13: File Descriptor Content Description

The attribute byte (FD_ATT) contains the file permission bits. Bit 7 is set to indicate a directory file, bit 6 indicates a non-sharable file, bit 5 is public execute, bit 4 is public write, etc.

The segment list (FD_SEG) consists of up to 48 five byte entries that have the size and address of each block of storage used by the file in logical order. Each entry has a three byte logical sector number that specifies the beginning of the block and a two byte block size (in sectors). Unused segments must be zero.

The RBF file manager is responsible for maintaining the file pointer, logical end-of-file, etc., used by application software, and converting them to the logical disk sector number using the data in the segment list.

The user does not have to be concerned with physical sectors at all. OS-9 provides fast random access to data stored anywhere in the file. All the information required to map the logical file pointer to a physical sector number is packaged in the file descriptor sector. This makes OS-9's record-locking functions very efficient.

Segment Allocation

Each device descriptor module has a value called a "segment allocation size" (see Figure 14). This parameter specifies the minimum number of sectors to allocate to a new segment. The goal is to avoid a large number of tiny segments when a file is expanded. If your system uses a small number of large files, this number should be set to a relatively high value, and viceversa.

When a file is created, it initially has no data segments allocated to it. Write operations past the current end-of-file (the first write is always past the end-of-file) cause additional sectors to be allocated to the file. Subsequent expansions of the file are also generally made in minimum allocation increments.

NOTE: An attempt is made to expand the last segment used when possible rather than adding a new segment.

When the file is closed, if not all of the allocated sectors are used, the segment will be truncated and any unused sectors deallocated in the bitmap. This strategy does not work very well for random-access data bases that expand frequently by only a few records. The segment list rapidly fills up with small segments. A provision has been added to prevent this from being a problem.

If a file (opened in write or update mode) is closed when it is not at end of file, **the last segment of the file will not be truncated.** In order to be effective, all programs that deal with the file in write or update mode must insure that they do not close the file while at end of file, or the file will lose any excess space it may have. The easiest way to insure this, is to do a seek(0) before closing the file. This method was chosen since random access files will frequently be at some other place than end of file, and sequential files are almost always at end of file when closed.

Directory File Format

Directory files have the same physical structure as other files with one exception. RBF must impose a convention for the logical contents of a directory file.

A directory file consists of an integral number of 32-byte entries. The end of the directory is indicated by the normal end-of-file. Each entry consists of a field for the file name and a field for the address of the file.

The file name field (DIR_NM) is 28 bytes long (bytes 0-27 of the entry) and has the sign bit of the last character of the file name set. The first byte is set to zero to indicate a deleted or unused entry. The address field (DIR_FD) is 3 bytes long (bytes 29-31 of the entry) and is the LSN of the file's FD sector. Byte 28 is not used and must be zero.

RAW PHYSICAL I/O ON DISK-TYPE DEVICES

An entire disk can be opened as one logical file. This allows any byte(s) or sector(s) to be accessed by physical address without regard to the normal file system. This feature is provided for diagnostic and utility programs that must be able to read and write to ordinarily non-accessible disk sectors.

A device is opened for physical I/O by appending the character "@" to the device name. For example, the device "/d2" can be opened for raw physical I/O under the pathlist "/d2@".

Standard open, close, read, write and seek system calls are used for physical I/O. A seek system call positions the file pointer to the actual disk physical address of any byte. To read a specific sector, perform a seek to the address computed by multiplying the LSN by 256. For example, to read physical disk sector 3, a seek is performed to address 768 (256*3) followed by a read system call, requesting 256 bytes.

If the number of tracks per sector of the disk is known or read from the Identification Sector, any track/sector address can be readily converted to a byte address for physical I/O.

WARNING: Improper physical I/O operations can corrupt the file system. Take great care when writing to a raw device. Physical I/O calls also bypass the file security system. For this reason, only super users are allowed to open the raw device for write permit. Non-super users are only permitted to read the identification sector (LSN 0) and the allocation bitmap. Attempts to read past this return an end-of-file error.

RECORD LOCKING

Record locking is a general term that refers to mechanisms that are designed to preserve the integrity of files that can be accessed by more than one user or process. OS-9 record locking is designed to be as invisible as possible to application programs. Most programs may be written without special concern for multi-user activity.

Simply stated, record locking involves:

1. Recognizing when a process is trying to read a record that another process may be modifying.

2. Deferring the read request until the record is "safe".

This is referred to as conflict detection and prevention. RBF record locking also handles non-sharable files and deadlock detection.

Record Locking and Unlocking

Conflict detection must determine when a record is in the process of being updated. RBF provides true record locking on a byte basis. A typical record update sequence is:

```
OS9 I$Read   program reads record        RECORD IS LOCKED
   .
   .         program updates record
   .
OS9 I$Seek   reposition to record
OS9 I$Write  record is rewritten         RECORD IS RELEASED
```

When a file is opened in update mode, ANY read will cause the record to be locked out because RBF does not know in advance if the record will be updated. The record remains locked until the next Read, Write or Close occurs. Reading files that are opened in read or execute modes does not cause record locking to occur because records can not be updated in these two modes.

A subtle but nasty problem exists for programs that interrogate a data base and occasionally update its data. When a user looks up a particular record, the record could be locked out indefinitely if the program neglects to release it. The problem is characteristic of record locking systems and can be avoided by careful programming.

It should be noted that only one portion of a file may be locked out at one time. If an application requires more than one record to be locked out, multiple paths to the same file may be opened each having its own record locked out. RBF will notice that the same process owns both paths and will keep them from locking each other out. Alternately, the entire file may be locked out, the records updated and the file released.

Non-Sharable Files

File locking may be used when an entire file is considered unsafe to be used by more than one user. Sometimes (rarely), it is necessary to create a file that can never be accessed by more than one process at a time (non-sharable). This is done by setting the single user (S) bit in the file's attribute byte. The bit can be set when the file is created, or later using the ATTR utility.

Once the single user bit has been set, only one process may open the file at a time. If another process attempts to open the file, an error (#253) will be returned.

More commonly, a file will need to be non-sharable only during the execution of a specific program. This is accomplished by opening the file with the single user bit set in the access mode parameter.

One example might be when the file is being sorted. If the file is opened as a non-sharable file, it will be treated exactly as though it had a single user attribute. If the file has already been opened by another process, an error (#253) will be returned.

A necessary quirk of non-sharable files is that they may be duplicated using the I$Dup system call, or inherited. A non-sharable file could therefore actually become accessible to more than one process at a time. Non-sharable only means that the file may be opened once. It is usually a very bad idea to have two processes actively using any disk file through the same (inherited) path.

End of File Lock

An EOF lock occurs when a user reads or writes data at the end of file. The user keeps the end of file locked until a read or write is performed that is not at the end of the file. EOF Lock is the only case that a write call automatically causes any of the file to be locked out. This avoids problems that could otherwise occur when two users want to simultaneously extend a file.

An interesting and extremely useful side effect occurs when a program creates a file for sequential output. As soon as the file is created, EOF Lock is gained, and no other process will be able to "pass" the writer in processing the file.

For example, if an assembly listing is redirected to a disk file, a spooler utility can open and begin listing the file before the assembler has written even the first line of output. Record locking will always keep the spooler "one step behind" the assembler, making the listing come out as desired.

DeadLock Detection

A deadlock can occur when two processes attempt to gain control of the same two disk areas at the same time. If each process gets one area (locking out the other process), both processes would be stuck permanently, waiting for a segment that can never become free. This situation is a general problem that is not restricted to any particular record locking method or operating system.

If this occurs, a deadlock error (#254) is returned to the process that caused it to be detected. It is easy to create programs that, when executed concurrently, generate lots of deadlock errors. The easiest way to avoid them is to access records of shared files in the same sequences in all processes that may be run simultaneously. For example, always read the index file before the data file, never the other way around.

When a deadlock error does occur, it is not sufficient for a program to simply re-try the operation "in error". If all processes used this strategy, none would ever succeed. It is necessary for at least one process to release its control over a requested segment for any to proceed.

RECORD LOCKING DETAILS FOR I/O FUNCTIONS

Open/Create: The most important guideline to follow when opening files is: **Do not open a file for update if you only intend to read.** Files open for read only will not cause records to be locked out, and they will generally help the system to run faster. If shared files are routinely opened for update on a multi-user system, users may sometimes become hopelessly record-locked for extended periods of time.

The special "@" file should be used in update mode with extreme care. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The "@" file is considered different from any other file, and therefore will only conform to record lockouts with other users of the "@" file.

Read/ReadLine Read and ReadLine cause records to be locked out only if the file is open in update mode. The locked out area includes all bytes starting with the current file pointer and extending for the number of bytes requested.

For example, if a ReadLine call is made for 256 bytes, exactly 256 bytes will be locked out, regardless of how many bytes are actually read before a carriage return is encount- ered. EOF Lock will occur if the bytes requested also includes the current end-of-file.

A record will remain locked until any of the following occur: another read is performed, a write is performed, the file is closed, or a record lock SetStat is issued. Releasing a record does not normally release EOF Lock. Any Read or Write of zero bytes will release any record lock, EOF lock or File Lock.

Write/WriteLine Write calls always release any record that has been locked out. In addition, a write of zero bytes releases EOF Lock and File Lock. Writing usually does not lock out any portion of the file unless it occurs at end of file when it will gain EOF Lock.

Seek Seek does not effect record locking.

SetStatus Two setstat codes have been included for the convenience of record locking. They are SS_Lock, for locking or releasing part of a file; and SS_Ticks, for setting the length of time a program is willing to wait for a locked record. See the I$SETSTT section (chapter 16) for a description of the codes.

FILE SECURITY

Each file has a group/user ID that identifies the file's owner. These are copied from the current process descriptor when the file is created. Usually a file's owner ID is never changed.

An attribute byte is also specified when a file is created. The file's attribute byte tells RBF in which modes a file may be accessed. Together with the file's owner ID, the attribute byte provides (some) file security.

The attribute byte has two sets of bits that indicate whether a file may be opened for read, write or execute by the owner or the public. In this context, the file's "owner" is any user having the same group ID as the file's creater. "Public" means any user with a different group ID.

Whenever a file is opened, access permissions are checked on all directories specified in the pathlist, as well as the file itself. If you do not have permission to read a directory, you may not read any files in that directory either.

Any super user (a user with group ID = 0) may access any file in the system. Files that are owned by the super user can not be accessed by any other user regardless of the group ID. Files containing modules that are owned by the super user must also be owned by the super user. If not, the modules contained within the file will not be loaded.

CAVEAT: Care should be taken by the system Manager when assigning group/user IDs. The RBF File Descriptor stores the group/user ID in a two byte field (FD_OWN). The group/user ID that resides in the password file is permitted 2 bytes for the group ID and two bytes for the user ID. RBF will only read the low order byte of both the group and user ID. Consequently a user with the ID of 256.512 will be mistaken for the super user by RBF.

RBF DEVICE DESCRIPTOR MODULES

This section describes the definitions of the initialization table contained in device descriptor modules for RBF-type devices. The table immediately follows the standard Device Descriptor Module Header fields (see Chapter 6 for full descriptions). A graphic representation of the table is shown in Figure 14. The size of the table is defined in the M$Opt field. For an example of an actual RBF descriptor, see Appendix B.

NAME **UTILIZATION**

PD_DTP **Device class**
(0=SCF 1=RBF 2=PIPE 3=SBF 4=NET)

PD_DRV **Drive number**
This location is used to associate a one byte integer with each drive that a controller will handle. Each controller's drives should be numbered 0 to n-1 (n = the maximum number of drives the controller can handle). This number also defines how many drive tables are required by the driver and RBF.

PD_STP **Step rate**
(Floppy disks) This location contains a code that sets the head stepping rate that will be used with the drive. The step rate should be set to the fastest value that the drive is capable of to reduce access time. Below are the values commonly used:

STEP CODE	5" Disks	8" Disks
0	30ms	15ms
1	20ms	10ms
2	12ms	6ms
3	6ms	3ms

PD_TYP **Disk Type device type (these parameters are format specific)**

```
bit   0 -- 000 = 5" floppy disk
           001 = 8" floppy disk

1,2,3,4 -- reserved

      5 -- 0 = Standard OS-9 format (track 0 single density)
           1 = Non-standard format  (track 0 double density)

      7 -- 0 = Floppy disk
           1 = Hard disk
```

NAME	UTILIZATION

PD_DN **Density byte ***

Density capabilities (Floppy disk only):

```
bit 0 -- 0 = Single bit density (FM)
         1 = Double bit density (MFM)

bit 1 -- 0 = Single track density (5", 48 TPI)
         1 = Double track density (5". 96 TPI)

bit 2 -- 1 = Reserved for Quad density
              (currently not supported)
```

PD_CYL **Cylinders-TrkOff number of cylinders (TRACKS) ***

This is the number of cylinders per disk.

PD_SID **Heads or Sides ***

This indicates the number of heads for a hard disk (Heads) or the number of surfaces for a floppy disk (Sides).

PD_VFY **Verify or NoVerify**

0 = verify disk write 1 = no verification

Write verify operations are generally performed on floppy disks but not hard disks because of the lower soft error rate of hard disks.

PD_SCT **Default sectors/track ***

This is the number of sectors per track.

PD_TOS **Default Sectors/Track (Track 0) ***

This is the number of sectors per track for track 0. This may be different than PD_SCT (depending on specific disk format).

PD_SAS **Segment allocation size**

This value specifies the default minimum number of sectors to be allocated when a file is expanded.

PD_ILV **Sector interleave factor ***

Sectors are arranged on a disk in a certain sequential order (1, 2, 3, etc. 1, 3, 5, etc). The interleave factor determines the arrangement. For example, if the interleave factor is 2, the sectors would be arranged by 2's (1, 3, 5, etc) starting at the base sector (see Sectoffs).

*** These parameters are format specific.**

NAME	UTILIZATION

PD_TFM **DMA transfer mode**
Direct Memory Access. This is hardware specific. If available the byte can be set for use of DMA mode. DMA requires only a single interrupt for each block of characters transferred in an I/O operation. It is much faster than methods that interrupt for each character transferred.

PD_TOffs **Track base offset ***
This is the offset to the first accessable track number. Because Track 0 is often a different density, Track 0 is sometimes not used as the base track.

PD_SOffs **Sector base offset ***
This is the offset to the first accessable sector number. Sector 0 is sometimes not the base sector.

PD_SSize **Sector Size**
This is the sector size in bytes. The default sector size is 256 bytes. PD_SSize is currently not used.

PD_Cntl **Control Word**
This is the format control word. It may currently contain the following:

```
bit 0 clear = format enable
bit 0 set   = format inhibit
bit 1 set   = multi-sector I/O capability
bit 2-7 reserved for future use
```

PD_Trys **Number of Tries**
This is the number of times a device will try to access a disk before returning an error. Currently, only two values are permitted:

```
0 = default (a driver will try several times to access a
    disk before returning an error; this is driver depen-
    dent)

1 = one try (no retries)
```

Any other value will allow the default number of retries. In future implementation, this value will represent the number of tries that will be made.

*** These parameters are format specific.**

NAME	UTILIZATION
PD_LUN	**Logical Unit Number of SCSI Drive** This number is the value which will be used in the SCSI command block to identify the drive to the controller. This may be different from the PD_DRV number, so that the drive tables need not have unused entries. For example: If the SCSI unit number is 7, the PD_DRV value could be 0. Consequently only one entry would be needed in the drive table instead of 8 (if PD_DRV is 7).
PD_WPC	**First Cylinder to Use Write Precompensation** This is the number used to determine at which cylinder to begin write precompensation.
PD_RWR	**First Cylinder to Use Reduced Write Current** This is the number used to determine at which cylinder to begin reduced write current.
PD_Park	**Cylinder Used to Park Head** This is the cylinder at which to park the hard disk's head, when the drive is to be shut down.
PD_LSNOffs	**Logical Sector Offset** This is the offset to be used when accessing a partitioned SCSI drive.
PD_TotCyls	**Total Cylinders On Device** This value is the actual number of cylinders on a partitioned drive. It is used by the driver, so that the drive may be correctly initialized.
PD_CtrlrID	**SCSI Controller ID** This is the ID number of the controller attached to the drive. The driver uses this number when communicating with the controller.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library, "sys.l".

Offset		Usage
$48	PD_DTP	Device Class
$49	PD_DRV	Drive Number
$4A	PD_STP	Step Rate
$4B	PD_TYP	Device Type
$4C	PD_DNS	Density
$4D		Reserved
$4E	PD_CYL	# of Cylinders
$50	PD_SID	# of Heads/Sides
$51	PD_VFY	Disk Write Verification
$52	PD_SCT	Default Sectors/Track
$54	PD_TOS	Default Sectors/Track 0
$56	PD_SAS	Segment Allocation Size
$58	PD_ILV	Sector Interleave Factor
$59	PD_TFM	DMA Transfer Mode
$5A	PD_TOffs	Track Base Offset

Offset		Usage
$5B	PD_SOffs	Sector Base Offset
$5C	PD_SSize	Sector Size (in bytes)
$5E	PD_Cntl	Control Word 0 = format enable 1 = format inhibit
$60	PD_Trys	# of Tries 1 = no retry
$61	PD_LUN	SCSI Unit Number of Drive
$62	PD_WPC	Cylinder to Begin Write Precompensation
$64	PD_RWR	Cylinder to Begin Reduced Write Current
$66	PD_Park	Cylinder to Park Disk Head
$68	PD_LSNOffs	Logical Sector Offset
$6C	PD_TotCyls	# of Cylinders On Device
$6E	PD_CtrlrID	SCSI Controller ID

Figure 14: Initialization Table for RBF Device Descriptor Modules

RBF DEFINITIONS OF THE PATH DESCRIPTOR

The first 19 fields of the reserved section (PD_OPT) of the path descriptor used by RBF are copied directly from the device descriptor initialization table. These fields can be updated by using GetStat and SetStat system calls. The final 7 fields are not copied from the device descriptor module and can not be updated. The RBF Path descriptor option table is shown below.

For descriptions of the first 19 fields, see the previous section on RBF device descriptors. The final 7 fields are not copied from the device descriptor and can not be updated using GetStat or SetStat system calls. Their description follows:

NAME	UTILIZATION
PD_ATT	**File Attributes** (D S PE PW PR E W R)
PD_FD	**File Descriptor** The LSN (Logical Sector Number) of the file is written here.
PD_DFD	**Directory File Descriptor** The LSN of the file's directory is written here.
PD_DCP	**File's Directory Entry Pointer**
PD_NAME	**File Name**

Offset		Usage
$80	PD_DTP	Device Class
$81	PD_DRV	Drive Rate
$82	PD_STP	Step Rate
$83	PD_TYP	Device Type
$84	PD_DNS	Density
$85		Reserved
$86	PD_CYL	# of Cylinders
$88	PD_SID	# of Heads/Sides
$89	PD_VFY	Write Disk Verification
$8A	PD_SCT	Default Sector/Track
$8C	PD_TOS	Default Sector/Track 0
$8E	PD_SAS	Segment Allocation Size
$90	PD_ILV	Sector Interleave Factor
$91	PD_TFM	DMA Transfer Mode
$92	PD_TOffs	Track Base Offset
$93	PD_SOffs	Sector Base Offset
$94	PD_SSize	Sector Size (in bytes)

Offset		Usage
$96	PD_Cntl	Control Word
$98	PD_Trys	# of Tries
$99	PD_LUN	SCSI Unit Number of Drive
$9A	PD_WPC	Cylinder to Begin Write Precompen.
$9C	PD_RWR	Cylinder to Begin Begin Reduced Write Current
$9E	PD_Park	Cylinder to Park Disk Head
$A0	PD_LSNOffs	Logical Sector Offset
$A4	PD_TotCyls	# of Cylinders On Device
$A6	PD_CtrlrID	SCSI Controller ID
$A7		Reserved
$B5	PD_ATT	File Attributes
$B6	PD_FD	File Descriptor LSN
$BA	PD_DFD	Directory File Descriptor LSN
$BE	PD_DCD	File Directory Entry Pointer
$C6		Reserved
$E0	PD_NAME	File Name

Figure 15: Option Table For RBF Path Descriptor

RBF DRIVERS

RBF-type device drivers are designed to support any random access storage device which reads and writes data in fixed size blocks (for example, disks or bubble memories).

OS-9 reads and writes in standard 256 byte sectors. The file manager takes care of all file system processing and passes the driver a 256-byte data buffer and a logical sector number (LSN) for each read or write operation.

Read calls to the driver initiate the sector read operation (and a prior "seek" operation if required). For interrupt driven systems, the controller will generate an interrupt when the data has been read into the buffer. The driver must suspend itself until the interrupt occurs. DMA (Direct Memory Access) operation is preferred if available.

Write calls to the driver initiates the sector write operation (and a prior "seek" operation if required). For interrupt driven systems, the controller generates an interrupt when the data has been written from the buffer onto the disk. The driver must suspend itself until the interrupt occurs. DMA operation is preferred if available. If the "verify" flag is set in the path descriptor (PD_VFY), the sector should be read back and verified.

Drivers for hard disks are relatively simple because the driver typically works with an intelligent controller, and because the disk format is fixed. For example, most SASI (SCSI) type hard disk controllers directly accept OS-9's logical sector number as the physical sector address.

Floppy disk drivers are more complicated because they work with less capable disk controllers and often must handle a variety of disk sizes (3", 5", 8") and physical formats (density, number of sides, track spacing).

OS-9 uses an access system for floppy disks that attempts to automatically adapt to all formats the drives and controllers are physically capable of using. For example, a system that can read double-sided/double-density floppy disks can usually read and write single-sided/double density or double-sided/single-density disks.

Disk drivers keep a table in their static variable storage area that contains current track addresses and disk format information for each drive (unit). The track addresses are used for controllers that have explicit "seek" commands to determine if the head must be moved prior to a read or write operation. The format data part of each table entry is used to select density, number of sides, etc.

The INIT routine obtains some initialization data from the device descriptor module. Each disk media has similar format information recorded on LSN zero (the FORMAT utility puts it there). Whenever sector zero of a floppy disk is read, the drive's table entry is updated with the information actually read. This is how the driver automatically adapts to different disk formats.

Initialization of the table must occur prior to access of any other sector on the drive.

RBF Device Driver Storage Definitions

RBF type device driver modules contain a package of subroutines that perform sector oriented I/O to or from a specific hardware controller. Because these modules are re-entrant, one "copy" of the module can simultaneously run several identical I/O controllers.

The kernel will allocate a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header (M$Mem). Some of this storage area is required by the kernel and RBF. The device driver may use the remainder in any manner. Information on device driver static storage required by the operating system can be found in the "rbfstat.a" and "drvstat.a" DEFS files. Static storage is used as follows:

NAME	UTILIZATION
V_PORT	**Device base port address.**
V_LPRC	**Last active process ID.** This contains the process ID is the last process to use the device. While this field is required for all device descriptors by the kernel, it is not used by RBF.
V_BUSY	**Current active process.** This contains the process ID of the process currently using the device. (0 = not busy)
V_WAKE	**Process ID to awaken.** This contains the process ID of any process that is waiting for the device to complete I/O (0 = no process waiting). Maintained by device driver.
V_PATHS	**Linked List of Open Paths.** This is a singly-linked list of all paths currently open on this device. It is maintained by the kernel.
V_NDRV	**Number of drives.** This contains the number of drives that the controller can use. It is defined by the device driver as the maximum number of drives that the controller can work with. RBF will assume that there is a drive table for each drive.
	Drive Tables This contains one table per drive that the controller will handle. RBF will assume there are as many tables as specified in V_NDRV.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library, "sys.l".

Offset		Usage
$00	V_PORT	Device base port address
$04	V_LPRC	Last active process ID
$06	V_BUSY	Current active process
$08	V_WAKE	Process ID to awaken
$0A	V_PATHS	Linked List of Open Paths
$2E	V_NDRV	Number of Drives
$2F		RESERVED
$36		Drive Tables

Figure 16: RBF Static Storage Allocation

Device Driver Tables

After the driver INIT routine has been called, RBF will request the driver to read the identification sector (LSN 0) from the drive. At this time the driver must initialize the corresponding drive table. It does this by copying the first 21 bytes of sector 0 (through DD_RES) into the appropriate table. The format of each drive table is given below:

Offset		Usage
$00	DD_TOT	Total Number of Sectors
$03	DD_TKS	Track Size (in sectors)
$04	DD_MAP	# of Bytes In Allocation Map
$06	DD_BIT	# of sectors/bit (cluster size)
$08	DD_DIR	LSN of Root Directory FD
$0B	DD_OWN	Owner ID
$0D	DD_ATT	Attributes
$0E	DD_DSK	Disk ID
$10	DD_FMT	Disk Format: Density/Sides
$11	DD_SPT	Sectors/Track
$13		Reserved
$16	V_TRAK	Current Track Number

Offset		Usage
$18	V_FileHd	Open File List For Disk
$1C	V_DiskID	Disk ID
$1E	V_BMapSz	Bitmap Size
$20	V_MapSct	Lowest Bitmap Byte To Search
$22	V_BMB	Bitmap In Use Flag
$24	V_ScZero	Pointer To Sector 0
$28	V_ZeroRd	Sector 0 Read Flag
$29	V_Init	Drive Initialized Flag
$2A	V_Resbit	Reserved Bitmap Sector Number
$2C	V_SoftEr	# Of Recoverable Errors
$30	V_HardEr	# Of Non-Recoverable Errors
$34		Reserved (32 bytes)

Figure 17: RBF Device Driver Table Format: there must be as many tables as were specified in NDRV

RBF Device Driver Subroutines

As with all device drivers, RBF device drivers use a standard executable memory module format with a module type of "Drivr" (code $E0).

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of a corresponding subroutine. The branch table is as follows:

```
ENTRY   dc.w  INIT      initialize device
        dc.w  READ      reads character
        dc.w  WRITE     writes character
        dc.w  GETSTA    gets device status
        dc.w  SETSTA    sets device status
        dc.w  TERM      terminates device
        dc.w  TRAP      handles illegal exception
```

Each subroutine should exit with the condition code register carry bit cleared if no error occurred. Otherwise the carry bit should be set and an appropriate error code returned in d1.w. The following pages give a description of each subroutine.

The TRAP entry should be defined. It should be set to an offset to exception handling code. It should be set to zero if no exception handler is available. This entry point is currently not used by the kernal. However, in future revisions this will be accessed.

The following pages give a description of each subroutine.

NAME: INIT

INPUT: (a1) = address of the device descriptor module
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system globel data pointer

OUTPUT: None

ERROR OUTPUT: cc = carry bit set
d1.w = error code

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE AREA

The INIT routine must:

1. Initialize the device's permanent storage. This minimally consists of:

   A. Initializing V_NDRV to the number of drives that the controller will work with.

   B. Initializing DD_TOT in the drive table to a non-zero value so that sector zero may be read or written to.

   C. Initializing V_TRAK to $FF so that the first seek will find track zero.

2. Initialize device control registers (enable interrupts if necessary).

3. Place the IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.

NOTE: Prior to being called, the device permanent storage will be cleared (set to zero) except for V_PORT which will contain the device address. The driver should initialize each drive table appropriately for the type of disk the driver expects to be used on the corresponding drive.

NAME: READ

INPUT:
```
d0.l = number of contiguous sectors to read
d2.l = disk logical sector number to read
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

OUTPUT: sector(s) returned in the sector buffer

ERROR OUTPUT:
```
cc = carry bit set
d1.w = Appropriate error code
```

FUNCTION: READ SECTOR(S)

The READ routine must:

1. Get the sector buffer address from PD_BUF in the path descriptor.
2. Verify the drive number from PD_DRV in the path descriptor.
3. Compute the physical disk address from the logical sector number.
4. Seek to the physical track requested.
5. Read sector(s) from the disk into the sector buffer.
6. Copy V_BUSY to V_WAKE. The driver then goes to sleep and waits for the I/O to complete (the IRQ service routine is responsible for sending a wake up signal and clearing V_WAKE). After awakening, it must test V_WAKE to see if it is clear. If not, it goes back to sleep.

If the disk controller can not be interrupt driven it will be necessary to perform programmed I/O.

NOTE: Whenever logical sector zero is read, the first part of it must be copied into the appropriate drive table. PD_DTB contains a pointer to the proper drive table entry. The number of bytes to copy is DD_SIZ.

If bit number 1 in the PD_Cntl field is clear, RBF will only request one sector READS. If the bit is set, RBF may request up to 255 contiguous sectors to be read.

NAME: WRITE

INPUT:
```
d0.l = number of contiguous sectors to write
d2.l = disk logical sector number
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

OUTPUT: The sector buffer is written to disk

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: WRITE SECTOR(S)

The WRITE routine must:

1. Get the sector buffer address from PD_BUF in the path descriptor.
2. Verify the drive number from PD_DRV in the path descriptor.
3. Compute the physical disk address from the logical sector number.
4. Seek to the physical track requested.
5. Write sector buffer(s) to the disk.
6. Copy V_BUSY to V_WAKE. The driver then goes to sleep and waits for the I/O to complete (the IRQ service routine is responsible for sending the wakeup signal and clearing V_WAKE). Test V_WAKE after awakening, to see if it is clear. If not, then the driver goes back to sleep.
7. If PD_VFY in the path descriptor is equal to zero, read the sector back and verify that it is written correctly. It is recommended that the compare loop be as short as possible to keep the necessary sector interleave value to a minimum.

If the disk controller can not be interrupt-driven, it will be necessary to perform a programmed I/O transfer.

If bit number 1 in the PD_Cntl field is clear, RBF will only request one sector WRITES. If the bit is set, RBF may request up to 255 contiguous sectors to be written.

NAME: GETSTAT/SETSTAT

INPUT:
```
d0.w = status code
(a1) = address of the path descriptor
(a2) = address of the device static storage area
(a4) = process descriptor pointer
(a5) = caller's register stack pointer
(a6) = system global data storage pointer
```

OUTPUT: Depends on the function code.

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: GET/SET DEVICE STATUS

These routines are wild card calls used to get (set) the device's operating parameters as specified for the OS9 I$GetStt and I$SetStt service requests.

It may be necessary to examine or change the register stack which contains the values of MPU registers at the time the I$GetStt or I$SetStt service request was made.

Typical RBF drivers have routines to handle the "SS_WTrk" and "SS_Reset" SetStat codes. Usually all GetStat codes and other SetStat codes return with an "E$UnkSvc" (UnKnown Service Request) error.

NAME: TERMINATE

INPUT:
```
(a1) = address of the device descriptor module
(a2) = Address of device static storage area
(a6) = OS-9 system global static storage
```

OUTPUT: None

ERROR OUTPUT: None

FUNCTION: TERMINATE DEVICE

This routine is called when a device is no longer in use in the system. This is defined as when the link count of its device table entry becomes zero (see I$Attach and I$Detach).

The TERM routine must:

1. Wait until any pending I/O has completed.
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.

NAME: IRQ SERVICE ROUTINE

INPUT:
```
(a2) = static storage address
(a3) = port address
(a6) = system global static storage
```

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device driver module branch table and is not called directly by RBF, it is a key routine in interrupt-driven device drivers. Its function is to:

1. Poll the device. If the interrupt is not caused by this device, the carry bit must be returned set with an RTS instruction as quickly as possible.

2. Service device interrupts.

3. Send a wake up signal to the process whose process ID is in V.WAKE, when the I/O is complete. Also, clear V_WAKE as a flag to the mainline program that the IRQ has indeed occurred.

4. When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.

NOTE: IRQ service routines may destroy the following registers only: **d0, d1, a0, a2, a3** and **a6**. All other registers used must be preserved or unpredictable system errors (i.e. system crashes) will occur.

SEQUENTIAL CHARACTER FILE MANAGER

SCF DESCRIPTION

The Sequential Character File Manager (SCF) is the OS-9 file manager module that supports devices which operate on a character by character basis: terminals, printers and modems. SCF can handle any number or type of character oriented devices. SCF is a reentrant subroutine package called for I/O service requests to SCF-type devices. It includes the extensive input and output editing functions that are typical of line-oriented operations such as backspace, line delete, repeat line, auto line feed, screen pause and return delay padding.

SCF LINE EDITING

The I$Read and I$Write service requests to SCF-type devices pass data to/from the device without any modification. Specifically, carriage returns are not automatically followed by line feeds or nulls, and the high order bits are passed as sent/received. If X-on and X-off are enabled, these characters are intercepted by the device driver and not processed by SCF.

The I$ReadLn and I$WritLn service requests to SCF-type devices perform full line editing of all functions enabled for the particular device.

These functions are initialized when a path is first opened by copying the option table from the device descriptor associated with that device into the path descriptor. They may be altered afterwards by assembly language programs using the I$SetStt and I$GetStt service requests or from the keyboard using TMODE.

SCF DEVICE DESCRIPTOR MODULES

Device descriptor modules for SCF-type devices contain the device address and an initialization table which defines initial values for the I/O editing features, as listed below. The initialization table immediately follows the standard Device Descriptor Module Header fields (see chapter 6 for full descriptions). The size of the table is defined in the M$Opt field. The initialization table is graphically shown in Figure 18 and the following table. See Appendix B for an example SCF device descriptor.

NOTE: It is possible to change or disable most of these special editing functions by changing the corresponding control character in the path descriptor. This can be done with the I$SetStt service request or by the TMODE utility. A more permanent solution may be to change the corresponding control character value in the device descriptor module. Device descriptors may be easily changed using the XMODE utility.

NAME	UTILIZATION

PD_DTP **Device class**
(0=SCF 1=RBF 2=PIPE 3=SBF 4=NET)

PD_UPC **Letter case**
If PD_UPC is not equal to zero, then input or output characters in the range "a..z" are made "A..Z"

PD_BSO **Destructive Backspace**
If PD_BSO is zero when a backspace character is input, SCF will echo PD_BSE (backspace echo character). If PD_BSO is non-zero, SCF will echo PD_BSE, space, PD_BSE.

PD_DLO **Delete**
If PD_DLO is zero, SCF will delete by backspace-erasing over the line. If PD_DLO is unequal to zero, SCF will delete by echoing a carriage return/line feed.

PD_EKO **Echo**
If PD_EKO is not zero, then all input bytes are echoed, except undefined control characters which are printed as a "." If PD_EKO is zero, input characters are not echoed.

PD_ALF **Automatic line feed**
If PD_ALF is not zero, then carriage returns are automatically followed by line feeds.

PD_NUL **End of line null count**
PD_NUL is a count of the number of NULL padding bytes (always $00) to be sent after a CR/LF character.

PD_PAU **End of page pause**
If PD_PAU is non-zero, an auto page pause will occur upon reaching a full screen of output. See PD_PAG for setting page length.

PD_PAG **Page length**
This contains the number of lines per screen (or page).

PD_BSP **Backspace "input" character**
This is the input character recognized as backspace. See also PD_BSE and PD_BSO.

NAME	UTILIZATION

PD_DEL **Delete line character**
This is the input character recognized as the delete line function. See also PD_DLO

PD_EOR **End of record character**
The PD_EOR character is the last character on each line entered (I$ReadLn). An output line is terminated (I$WritLn) when this character is sent. Normally PD_EOR should be set to $0D. Warning: If it is set to zero, SCF's ReadLn will NEVER terminate, unless an EOF occurs.

PD_EOF **End of file character**
PD_EOF defines the end of file character. SCF will return an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input. It can be disabled by setting its value to zero.

PD_RPR **Reprint line character**
When this character is input, SCF (I$ReadLn) will reprint the current input line. A carriage return is also inserted in the input buffer for PD_DUP (see below). This makes correcting typing errors more convenient.

PD_DUP **Duplicate last line character**
If this character is input, SCF (I$ReadLn) will duplicate whatever is in the input buffer through the first "PD_EOR" character. Normally, this will be the previous line typed.

PD_PSC **Pause character**
If this character is typed during output, output is suspended before the next end-of-line. This will also delete any "type ahead" input for I$ReadLn.

PD_INT **Keyboard interrupt character**
If PD_INT is input, a keyboard interrupt signal is sent to the last user of this path. It will terminate the current I/O request (if any) with an error identical to the keyboard interrupt signal code. PD_INT normally is set to a control-C character.

PD_QUT **Keyboard abort character**
When this character is input, a keyboard abort signal is sent to the last user of this path. It will terminate the current I/O request (if any) with an error code identical to the keyboard interrupt signal code. This value is normally a control-E character.

NAME	UTILIZATION

PD_BSE **Backspace "output" character (echo character)**
This is the backspace character to echo when PD_BSP is input. See PD_BSP and PD_BSO.

PD_OVF **Line overflow character**
If I$ReadLn has satisfied its input byte count, SCF ignores any further input characters until an end-of-line (PD_EOR) character is received. It echoes the PD_OVF character for each byte ignored. PD_OVF is usually set to the terminal's bell character.

PD_PAR **Parity code, number of stop bits & bits/character**
Bits 0 and 1 set the parity as follows:

```
0 = no parity
1 = odd parity
3 = even parity
```

Bits 2 and 3 set the number of bits per character as follows:

```
3 = 5 bits/character
2 = 6 bits/character
1 = 7 bits/character
0 = 8 bits/character
```

Bits 4 and 5 set the number of stop bits as follows:

```
0 = 1 stop bit
1 = 1 1/2 stop bits
2 = 2 stop bits
```

Bits 6 and 7 are reserved.

PD_BAU **Software adjustable baud rate**
This one byte field sets the baud rate as follows:

```
0 = 50 baud      6 = 600 baud     C = 4800 baud
1 = 75 baud      7 = 1200 baud    D = 7200 baud
2 = 110 baud     8 = 1800 baud    E = 9600 baud
3 = 134.5 baud   9 = 2000 baud    F = 19200 baud
4 = 150 baud     A = 2400 baud   FF = External
5 = 300 baud     B = 3600 baud
```

NAME	UTILIZATION
PD_D2P	**Offset to output device descriptor name string** SCF sends output to the device named in this string. Input comes from the device named by the M$PDev field. This permits two separate devices (i.e., a keyboard and video display) to be one logical device. Usually PD_D2P refers to the name of the same device descriptor it appears in.
PD_XON	**X-on character** See PD_XOFF below.
PD_XOFF	**X-off character** When this character is received, output from an SCF device is immediately stopped until an X-on character is received. This is required for software handshaking for some devices.
PD_Tab	**Tab character** In I$WritLn calls, SCF will expand this character into spaces to make tab stops at column intervals specified by PD_Tabs. NOTE: SCF does not know the effect of control characters on particular terminals. It can expand tabs incorrectly if they are used.
PD_Tabs	**Tab field size** See PD_Tab.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

Offset		Usage
$48	PD_DTP	Device Type
$49	PD_UPC	Upper Case Lock
$4A	PD_BSO	Backspace Option
$4B	PD_DLO	Delete Line Character
$4C	PD_EKO	Echo
$4D	PD_ALF	Automatic Line Feed
$4E	PD_NUL	End Of Line Null Count
$4F	PD_PAU	End Of Page Pause
$50	PD_PAG	Page Length
$51	PD_BSP	Backspace Input Character
$52	PD_DEL	Delete Line Character
$53	PD_EOR	End Of Record Character
$54	PD_EOF	End Of File Character
$55	PD_RPR	Reprint Line Character

Offset		Usage
$56	PD_DUP	Duplicate Line Character
$57	PD_PSC	Pause Character
$58	PD_INT	Keyboard Interrupt Character
$59	PD_QUT	Keyboard Abort Character
$5A	PD_BSE	Basckspace Output
$5B	PD_OVF	Line Overflow Character (bell)
$5C	PD_PAR	Parity Code # of Stop Bits and # of Bits/Character
$5D	PD_BAU	Adjustable Baud Rate
$5E	PD_D2P	Offset To Output Device Name
$60	PD_XON	X-ON Character
$61	PD_XOFF	X-OFF Character
$62	PD_TAB	Tab Character
$63	PD_TABS	Tab Column Width

Figure 18: Device Descriptor Initialization Table

SCF DEFINITIONS OF THE PATH DESCRIPTOR

The first 27 fields of the reserved section (PD_OPT) of the SCF path descriptor are copied directly from the SCF device descriptor initialization table. The table is shown in Figure 19.

These fields can be changed or disabled with the I$SetStt Service request, or the TMODE utility. A more permanent change may be to change the device descriptor table using the XMODE utility.

The SCF editing functions may be disabled by setting the corresponding control character value to zero. For example, by setting PD_INT to zero, there would be no "keyboard interrupt" character.

NOTE: Full definitions for the fields copied from the device descriptor are available in the previous section.

NOTE: The term "offset" in Figure 19 refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

Offset		Usage
$80	PD_DTP	Device Type
$81	PD_UPC	Upper Case Lock
$82	PD_BSO	Backspace Option
$83	PD_DLO	Delete Line Character
$84	PD_EKO	Echo
$85	PD_ALF	Automatic Line Feed
$86	PD_NUL	End Of Line Null Count
$87	PD_PAU	End Of Page Pause
$88	PD_PAG	Page Length
$89	PD_BSP	Backspace Input Character
$8A	PD_DEL	Delete Line Character
$8B	PD_EOR	End Of Record Character
$8C	PD_EOF	End Of File Character
$8D	PD_RPR	Reprint Line Character
$8E	PD_DUP	Duplicate Line Character
$8F	PD_PSC	Pause Character

Offset		Usage
$90	PD_INT	Keyboard Interrupt Character
$91	PD_QUT	Keyboard Abort Character
$92	PD_BSE	Basckspace Output
$93	PD_OVF	Line Overflow Character (bell)
$94	PD_PAR	Parity Code # of Stop Bits and # of Bits/Character
$95	PD_BAU	Adjustable Baud Rate
$96	PD_D2P	Offset To Output Device Name
$98	PD_XON	X-ON Character
$99	PD_XOFF	X-OFF Character
$9A	PD_TAB	Tab Character
$9B	PD_TABS	Tab Column Width
$9C		Reserved
$A0	PD_COL	Current Column
$A2	PD_ERR	Most Recent Error Status
$A3		Reserved

Figure 19: Path Descriptor Module Option Table For I/O Editing

SCF DRIVERS

SCF-type device drivers support I/O devices that read and write data a single character at a time, such as serial devices.

Generally, the input data (usually from a keyboard) is buffered. Each READ system call returns a single character at a time from the circular FIFO buffer. If the buffer is empty when a READ occurs, the driver must generate interrupts and suspend the calling process until an input character is received.

The GetStat system call permits an application program to test if the buffer contains any data. By checking first, the program will not be suspended if no data is available.

The driver may optionally handle full input buffer conditions using XON/XOFF or similar protocols. The input routine must also handle the special pause, abort and quit control characters. All other control characters (such as backspace, line delete, etc.) are handled at the file manager level.

The output data may or may not be buffered, depending on the physical characteristics of the output device. If the device is a memory-mapped video display driven by the main CPU, buffering and interrupts are not needed.

If the device is a serial interface, buffering and interrupts should be used. Each WRITE call passes a single output character to the driver which is placed in a circular FIFO output buffer. The output interrupt routine takes output characters from this buffer. If the buffer is full after a WRITE call, the driver should suspend the calling process until the buffer empties sufficiently.

SCF Device Driver Storage Definitions

SCF device driver modules contain a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical I/O controllers.

An individual static storage area is allocated for each copy of the device driver. The kernel determines that a new copy of the device driver is needed when an attach occurs for a device with a new port address.

The size of this storage area is given in the device driver module header (M$MEM). Some of this storage area is required by SCF. The device driver may use the remainder for variables and buffers.

The static storage required by SCF is defined in "scfstat.a" in the DEFS directory. It is usually included in the device static storage requirements by linking the file "scfstat.r" with the device driver relocatable file. SCF static storage is used as follows:

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

Offset		Usage
$00	V_PORT	Device base address
$04	V_LPRC	Last active process ID
$06	V_BUSY	Active process ID
$08	V_WAKE	Process ID to awaken
$0A	V_Paths	Linked list of open paths
$0E		Reserved
$2E	V_DEV2	Address of attached device static storage
$32	V_TYPE	Device type or parity
$33	V_LINE	Lines left until end of page
$34	V_PAUS	Pause request
$35	V_INTR	Keyboard interrupt character
$36	V_QUIT	Keyboard abort character
$37	V_PCHR	Pause character
$38	V_ERR	Error accumulator
$39	V_XON	X-on character
$3A	V_XOFF	X-off character
$3C		Reserved
$54	Device Driver Variables Begin Here	

Figure 20: Static Storage Allocation for SCF Device Drivers

NAME	UTILIZATION

V_PORT **Device base address**
This field is initialized by the kernel from the device port address.

V_LPRC **Last active process ID**
This contains the process ID of the last process to use the device. The IRQ service routine sends this process the proper signal when a "interrupt" or "quit" character is received.

V_BUSY **Active process ID (0 = not busy)**
This contains the process ID of the process currently using the device. This is used by SCF to prevent more than one process from using the device at the same time. V_BUSY is always equal to V_LPRC or 0.

V_WAKE **Process ID to reawaken**
This contains the process ID of any process that is waiting for the device to complete I/O (zero means there is no process waiting).

V_Paths **Linked list of open paths**
This is used by the kernel to determine if a non-sharable device is already in use.

V_DEV2 **Attached device static storage**
This contains the address of the ECHO (output) device's static storage area. Typically a device is it's own echo device. However, it may not be, as in the case of a keyboard and a memory mapped video display.

V_TYPE **Device type or parity**
This value is copied from PD_PAR in the path descriptor by SCF. It is typically used as a value to initialize the device control register, for parity, etc.

V_LINE **Lines left until end of page**
This contains the number of lines left until the end of the page. Paging is handled by SCF.

V_PAUS **Pause request**
This is a flag used to signal SCF that a pause character has been received. Setting its value to anything other than 0 will cause SCF to stop transmitting characters at the end of the next line. Device driver input routines must set V_PAUS in the ECHO device's static storage area. SCF will check this value in the ECHO device's static storage when output is sent.

NAME	UTILIZATION
V_INTR	**Keyboard interrupt character** This value is copied from PD_INT in the path descriptor by SCF.
V_QUIT	**Quit character** This value is copied from PD_QUT in the path descriptor by SCF.
V_PCHR	**Pause character** This value is copied from PD_PSC in the path descriptor by SCF.
V_ERR	**Error accumulator** This location is used to accumulate I/O errors. Typically, it is used by the IRQ service routine to record errors so that they may be reported later when SCF calls one of the device driver routines.
V_XON	**X-on character** This character is copied from the PD_XON field of the path descriptor. See V_XOFF below.
V_XOFF	**X-off character** This character is copied from the PD_XOFF field of the path descriptor. When an X-off character is received, the driver should immediately disable output interrupts and stop sending characters. Output interrupts are enabled only when the V_XON character is received. Both V_XON and V_XOFF are "eaten" by the device driver and NOT put into the circular FIFO buffer.

SCF DEVICE DRIVER SUBROUTINES

As with all device drivers, SCF device drivers use a standard executable memory module format with a module type of "device driver" **(Drivr)**.

The execution offset address in the module header points to a branch table that has seven entries. Each entry contains the offset of the corresponding subroutine. The entry table is as follows:

```
ENTRY    dc.w  INIT     initialize device
         dc.w  READ     read character
         dc.w  WRITE    write character
         dc.w  GETSTA   get device status
         dc.w  SETSTA   set device status
         dc.w  TERM     terminate device
         dc.w  TRAP     handles illegal exception
```

Each subroutine should exit with the condition code register Carry bit cleared if no error occurred. Otherwise, the Carry bit should be set and an appropriate error code returned in the least significant word of register d1.

The TRAP entry should be defined as the offset to exception handling code or zero if no handler is available. This entry point is currently not used by the kernal. However, in future revisions this will be accessed.

The following pages contain descriptions of each subroutine's functions and parameters.

NAME: INIT

INPUT:
```
(a1) = address of device descriptor module
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Error code
```

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE

The INIT routine must:

1. Initialize the device static storage.
2. Initialize the device control registers.
3. Place the driver IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.
4. Enable interrupts if necessary.

NOTE: Prior to being called, the device static storage will be cleared (set to zero) except for V_PORT which will contain the device port address. Do not initialize the portion of static storage used by SCF.

NAME: READ

INPUT:
```
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

OUTPUT: `d0.b = input character`

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Error code
```

FUNCTION: GET NEXT CHARACTER

This routine gets the next character from the input buffer. If there is no data ready, this routine copies its process ID from V_BUSY into V_WAKE and then uses the F$Sleep service request to put itself to sleep indefinitely.

When an input character is received, the IRQ service routine should put the data in the buffer. It then checks V_WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine sends a wakeup signal to the waiting process and clears V_WAKE.

NOTE: Data buffers for queueing data between the main driver and the IRQ service routine are NOT automatically allocated. They should be defined in the device's static storage area.

NAME: WRITE

INPUT:
```
d0.b = char to write
(a1) = address of the path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Error code
```

FUNCTION: OUTPUT A CHARACTER

This routine places a data byte into an output buffer and enables the device output interrupt. If the data buffer is already full, this routine should copy its process ID from V_BUSY into V_WAKE and then put itself to sleep.

When the IRQ service routine transmits a character and makes room for more data in the buffer, it checks V_WAKE to see if there is a process waiting for the device to complete I/O. If there is, it sends a wake up signal to that process and clears V_WAKE.

NOTE: This routine must ensure that output interrupts are enabled if necessary. After an interrupt is generated the IRQ service routine will continue to transmit data until the data buffer is empty, and then it should disable the device's "ready to transmit" interrupts.

NOTE: Data buffers or queues between the main driver and the IRQ routine used are defined in the device's static storage.

NAME: GETSTAT/SETSTAT

INPUT:
```
d0.w = function code
(a1) = address of path descriptor
(a2) = address of device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

OUTPUT: Depends upon function code

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Error code
```

FUNCTION: GET/SET DEVICE STATUS

These routines are a wild card calls used to get (set) the device parameters specified in the I$GetStt and I$SetStt service requests. Many SCF-type requests are handled by the kernel or SCF. Any codes not defined by them will be passed to the device driver.

In writing getstat/setstat codes, it may be necessary to examine or change the register stack which contains the values of the 68000 registers at the time the OS-9 service request was issued. The address of the register packet may be found in PD_RGS, which is located in the path descriptor.

If a status report is made to a unrecognized device driver, E$UnkSvc (Unknown Service Request) should be returned as an error.

NAME: TERMINATE

INPUT:
```
(a1) = device descriptor pointer
(a2) = ptr to device static storage
(a4) = process descriptor pointer
(a6) = system global data pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: TERMINATE DEVICE

This routine is called when a device is no longer in use. This is defined as when the link count of its device table entry becomes zero. It must perform the following:

1. Wait until the output buffer has been emptied (by the IRQ service routine).
2. Disable device interrupts.
3. Remove device from the IRQ polling list.

NAME: IRQ SERVICE ROUTINE

INPUT: (a2) = static storage
(a3) = port address
(a6) = system global static storage

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device drivers branch table and not called directly from SCF, it is an important routine in interrupt-driven device drivers. Its function is:

1. Poll the device. If the device did not cause the interrupt, exit immediately with an RTS instruction and the carry bit set. This section should be as fast as possible.

2. Service the device interrupts (receive data from device or send data to it). This routine should put its data into and get its data from buffers which are defined in the device static storage.

3. Wake up any process waiting for I/O to complete by checking to see if there is a process ID in V_WAKE (non-zero). If so, send a wakeup signal to that process and clear V_WAKE.

4. If the device is ready to send more data and the output buffer is empty, disable the device's "ready to transmit" interrupts.

5. If a pause character is received, set V_PAUS in the attached device static storage to a non-zero value. The address of the attached device static storage is in V_DEV2.

6. If a keyboard abort or interrupt character is received, signal the process in V_LPRC (last known process) if any.

When the IRQ service routine finishes servicing an interrupt, it must clear the carry and exit with an RTS instruction.

NOTE: IRQ service routines may destroy the following registers only: **d0, d1, a0, a2, a3** and **a6**. All other registers used must be preserved or unpredictable system errors (i.e. system crashes) will occur.

End of Chapter 8

NOTES

SEQUENTIAL BLOCK FILE MANAGER

SBF DESCRIPTION

The sequential Block File Manager (SBF) is a re-entrant subroutine package for I/O service requests to sequential block-oriented mass storage devices (tape systems). SBF can handle any number or type of such systems simultaneously.

TAPE I/O

SBF is designed to support both buffered and unbuffered I/O. It is capable of handling variable block sizes. If the PD_NumBlk field of the path descriptor is set to 0, unbuffered I/O is specified. If the field is set to a positive number, buffered I/O is used.

Unbuffered I/O

Unbuffered I/O is initiated by Read or Write service requests.

A Read request must specify a byte count greater than or equal to the size of the block used on the tape device. The block of data from the tape is read directly into the user's buffer. The byte count returned by the system call is the size of the block read.

Attempting to read a block with a byte count less than the tape block size will return a read error (#244).

Buffered I/O

All buffered I/O is initiated asyncronously by an auxiliary process created by SBF. SBF uses a "pool" of buffers to accomplish this. The number of buffers to be used is specified by the PD_NumBlk field of the path descriptor. The size of the buffer to be used is specified by the PD_BlkSiz field of the path descriptor.

Read requests will cause SBF to copy data from the buffer containing the block read. If the buffer is not yet available, SBF will allocate a new buffer and pass it to the auxiliary process. SBF then waits for the auxiliary process to return the buffer containing the next block. Multiple buffers (up to the number specified by PD_NumBlk) may be allocated, thus allowing SBF to be copying data from one buffer while the auxiliary process is reading data into others.

Write requests cause SBF to copy data into a buffer and return to the user immediately. When a buffer fills, SBF passes it to the auxiliary process for writing. If another buffer is required before the auxiliary process has had time to write the previous buffer, SBF allocates a new buffer and copies data to it. This allows SBF to be copying data into one buffer while the auxiliary process is writing from others.

End-Of-Tape Processing

There is no "end-of-tape" error on Read requests. Consequently, SBF requires an end of file mark to be present or the user process to handle the situation (i.e. knowing the size of the file or through the use of an end of data record).

Write requests will return a media full error (E$Full) when end-of-tape is reached. All prior writes will have completed; no other data may be written other than file marks after the end-of-tape has been reached.

SBF DEVICE DESCRIPTOR MODULES

This section describes the definitions of the initialization table contained in device descriptor modules for SBF-type devices. The table immediately follows the standard Device Descriptor Module Header fields (see Chapter 6 for full descriptions). A graphic representation of the table is shown in Figure 21. The size of the table is defined in the M$Opt field. For an example of an actual SBF descriptor, see Appendix B.

NOTE: The term "offset" refers to the location of a module field relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "usr.l".

Offset		Usage
$48	PD_DTP	Device Class
$49	PD_TDrv	Tape Drive Number
$4A	PD_SBF	Reserved
$4B	PD_NumBIK	Maximum Number of Blocks to Allocate
$4C	PD_BlkSiz	Size of Blocks Allocated
$50	PD_Prior	Driver Process Priority
$52	PD_Flags	Drive Capability Flags

Figure 21: Initialization Table for SBF Device Descriptor Modules

NAME	UTILIZATION

PD_DTP **Device class**
(0=SCF 1=RBF 2=PIPE 3=SBF 4=NET)

PD_TDrv **Tape Drive number**
This location is used to associate a one byte integer with each drive that a controller will handle. Each controller's drives should be numbered 0 to n-1 (n = the maximum number of drives the controller can handle). This number also defines how many drive tables are required by the driver and SBF.

PD_NumBlk **Number of Buffers/Blocks Used For Buffering**
This field specifies the maximum number of buffers to be allocated by SBF for use by the auxiliary process in buffered I/O. If this field is set to 0, unbuffered I/O is specified.

PD_BlkSiz **Size of Buffer/Block Used For I/O**
This field specifies the size of the buffer to be allocated by SBF. This buffer size is used when allocating multiple buffers used in buffered I/O.

PD_Prior **Driver Process Priority**
This is the priority at which SBF's auxiliary process will run. This value is used during initialization. Changing this value after initialization will have no affect.

PD_Flags **Drive Capability Flags**
This field specifies the capabilities of the tape controller used on the individual system. The flag definitions possible are:

```
(f_rest_b) bit 0 set = rewind on close flag
(f_offl_b) bit 1 set = offline on close flag
(f_eras_b) bit 2 set = erase to end of tape on close flag
```

This field is used by the SBF driver.

SBF DEFINITIONS OF THE PATH DESCRIPTOR

The reserved section (PD_OPT) of the path descriptor used by SBF is copied directly from the initialization table of the device descriptor. The following table is provided to show the offsets used in the path descriptor. For a full explanation of the path descriptor fields, refer to the previous pages.

NOTE: The term "offset" refers to the location of a module field relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "usr.l".

Offset		Usage
$80	PD_DTP	Device Class
$81	PD_TDrv	Tape Drive Number
$82	PD_SBF	Reserved
$83	PD_NumBlk	Maximum Number of Blocks Allocated
$84	PD_BlkSiz	Size of blocks Allocated
$88	PD_Prior	Driver Process Priority
$8A	PD_Flags	Drive Capability Flags

Fig

Figure 22: Option Table for SBF Path Descriptor Modules

SBF DRIVERS

SBF-type device drivers are designed to support any sequential storage device which reads and writes data in fixed size blocks (i.e tapes).

Because SBF is intended for sequentially accessed files, it does not support a directory structure or provide a byte oriented file positioning mechanism. Consequently, I$Makdir, I$ChgDir, I$Delete and I$Seek return the error E$UnkSvc.

SBF Device Driver Storage Definitions

SBF type device driver modules contain a package of subroutines that perform block oriented I/O to or from a specific hardware controller. Because these modules are re-entrant, one "copy" of the module can simultaneously run several identical I/O controllers.

The kernel will allocate a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header (M$Mem). Some of this storage area is required by the kernel and SBF. The device driver may use the remainder in any manner. SBF does not reserve any memory for drive tables. Each driver is responsible for reserving enough memory for the appropriate number of tables. Information on device driver static storage required by the operating system can be found in the "sbfstat.a" and "drvstat.a" DEFS files. Static storage is used as follows:

NAME	UTILIZATION
V_PORT	**Device base port address.**
V_LPRC	**Last active process ID.** This contains the process ID is the last process to use the device. While this field is required for all device descriptors by the kernel, it is not used by SBF.
V_BUSY	**Current active process.** This contains the process ID of the process currently using the device. (0 = not busy)
V_WAKE	**Process ID to awaken.** This contains the process ID of any process that is waiting for the device to complete I/O (0 = no process waiting). Maintained by device driver.

NAME	UTILIZATION
V_PATHS	**Linked List of Open Paths.** This is a singly-linked list of all paths currently open on this device. It is maintained by the kernel.
SBF_NDRV	**Number of drives.** This contains the number of drives that the controller can use. It is defined by the device driver as the maximum number of drives that the controller can work with. SBF will assume that there is a drive table for each drive.
SBF_Flag	**Driver Flags**
SBF_Drvr	**Driver Module Pointer**
SBF_DPrc	**Driver Process Pointer**
SBF_IPrc	**Interrupt Process Pointer**
	Drive Tables This contains one table per drive that the controller will handle. SBF will assume there are as many tables as specified in SBF_NDRV.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the static storage area. Offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library, "sys.l".

Offset		Usage
$00	V_PORT	Device base port address
$04	V_LPRC	Last active process ID
$06	V_BUSY	Current active process
$08	V_WAKE	Process ID to awaken
$0A	V_PATHS	Linked List of Open Paths
$2E		Reserved
$30	SBF_NDRV	Number of Drives
$31		Reserved
$32	SBF_Flag	Driver Flags
$34	SBF_Drvr	Driver Module Pointer
$38	SBF_DPrc	Driver Process Pointer
$3C	SBF_IPrc	Interrupt Process Pointer
$40		Reserved
$80		Drive Tables

Figure 23: SBF Static Storage Allocation

Device Driver Tables

The format of each drive table is given below:

Offset		Usage
$00	SBF_DFlg	Drive Flag
$02	SBF_NBuf	Buffer Count
$04	SBF_IBH	Pointer to Top of Input Buffer List
$08	SBF_IBT	Pointer to End of Input Buffer List
$0C	SBF_OBH	Pointer to Top of Output Buffer List
$10	SBF_OBT	Pointer to End of Output Buffer List
$14	SBF_Wait	Pointer to Waiting Process
$18	SBF_SErr	# of Recoverable Errors
$1C	SBF_HErr	# of Non-Recoverable Errors
$20		Reserved

Figure 24: SBF Device Driver Table Format: there must be as many tables as were specified in NDRV

SBF Device Driver Subroutines

As with all device drivers, SBF device drivers use a standard executable memory module format with a module type of "Drivr" (code $E0).

The execution offset address in the module header points to a branch table that has seven entries. Each entry is the offset of a corresponding subroutine. The branch table is as follows:

```
ENTRY   dc.w  INIT      initialize device
        dc.w  READ      reads character
        dc.w  WRITE     writes character
        dc.w  GETSTA    gets device status
        dc.w  SETSTA    sets device status
        dc.w  TERM      terminates device
        dc.w  TRAP      handles illegal exception
```

Each subroutine should exit with the condition code register carry bit cleared if no error occurred. Otherwise the carry bit should be set and an appropriate error code returned in d1.w. The following pages give a description of each subroutine.

The TRAP entry should be defined as the offset to exception handling code or zero if no handler is available. This entry point is currently not used by kenal. However, in future revisions this will be accessed.

The following pages give a description of each subroutine.

NAME: INIT

INPUT:
```
(a1) = address of the device descriptor module
(a2) = address of device static storage
(a6) = system global data pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = carry bit set
d1.w = error code
```

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE AREA

The INIT routine must:

1. Initialize the device's permanent storage. This minimally consists of initializing SBF_NDRV two'the number of drives that the controller will work with.

2. **Place the IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.**

3. **Initialize device control registers (enable interrupts if necessary).**

NOTE: Prior to being called, the device permanent storage will be cleared (set to zero) except for V_PORT which will contain the device address.

NAME: READ

INPUT:
```
d0.l = buffer size
(a0) = address of buffer
(a2) = address of device static storage
(a3) = drive table
(a4) = process descriptor pointer
(a6) = system global data storage pointer
```

OUTPUT: d1.l = block size read

ERROR OUTPUT:
```
cc = carry bit set
d1.w = Appropriate error code
```

FUNCTION: READ SECTOR(S)

The READ routine must:

1. Verify the drive number from PD_DRV in the path descriptor.
2. Initiate the read of block from the tape into the buffer.
3. Deactivate process pending completion of read.
4. After being reactivated by the IRQ service routine, return size of block read in d1.

NAME: WRITE

INPUT:
```
d0.l = buffer size
(a0) = address of buffer
(a2) = address of the device static storage area
(a3) = drive table
(a4) = process descriptor pointer
(a6) = system global data storage pointer
```

OUTPUT: The buffer is written to tape as one block

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: WRITE SECTOR(S)

The WRITE routine must:

1. Verify the drive number from PD_DRV in the path descriptor.
2. Initiate the write of buffer to the tape as one block.
3. Deactivate process pending completion of write.
4. After being reactivated by the IRQ service routine, return error status.

NAME: GETSTAT/SETSTAT

INPUT:
```
d0.w = status code
(a2) = address of the device static storage area
(a3) = drive table
(a4) = process descriptor pointer
(a6) = system global data storage pointer
```

OUTPUT: Depends on the function code.

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: GET/SET DEVICE STATUS

These routines are wild card calls used to get (set) the device's operating parameters as specified for the OS9 I$GetStt and I$SetStt service requests.

Typical SBF drivers have routines to handle the following SetStat codes:

```
SS_Reset:  rewind tape
SS_Feed:   Erase
SS_SQD:    Offline drive
SS_WFM:    Write tape mark(s)
SS_RFM:    Skip past tape mark(s)
SS_Skip:   Skip block(s)
```

Usually all GetStat codes and other SetStat codes return with an "E$UnkSvc" (UnKnown Service Request) error.

NAME: TERMINATE

INPUT:
```
(a1) = address of the device descriptor module
(a2) = Address of device static storage area
(a6) = OS-9 system global static storage
```

OUTPUT: None

ERROR OUTPUT: None

FUNCTION: TERMINATE DEVICE

This routine is called when a device is no longer in use in the system. This is defined as when the link count of its device table entry becomes zero (see I$Attach and I$Detach).

The TERM routine must:

1. Wait until any pending I/O has completed.
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.
4. Return driver process.

NAME: IRQ SERVICE ROUTINE

INPUT: (a2) = static storage address
(a6) = system global static storage

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device driver module branch table and is not called directly by SBF, it is a key routine in interrupt-driven device drivers. Its function is to:

1. Poll the device. If the interrupt is not caused by this device, the carry bit must be returned set with an RTS instruction as quickly as possible.

2. Service device interrupts.

3. Reactivate any process that was waiting for this interrupt.

4. When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.

NOTES

PIPEMAN: THE PIPE FILE MANAGER

Pipeman is the OS-9 file manager that supports interprocess communication through the use of "pipes." Pipes enable concurrently executing processes to communicate data: the output of one process (the writer) is read as input by a second process (the reader). Communication through pipes eliminates the need for an intermediate file to hold data.

Pipeman is a re-entrant subroutine package that is called for I/O service requests to a device named "/pipe." Even though no physical device is used in pipe communications, a driver must be specified in the pipe descriptor module. A "null" driver (a driver that does nothing) is used, but actually never gets called by Pipeman.

Pipes

A pipe is constructed as a first in first out (FIFO) buffer that usually contains 256 bytes. Typically, two processes share the pipe path: one writing and one reading. However, multiple processes can access the same pipe simultaneously. Pipeman coordinates the processes. The reader waits for the data to become available and the writer waits for the buffer to empty.

Pipes are generally thought of as a one way data path between two processes, but any number of processes can share a single path. A single pipe can even send data to itself. This might be used to simplify type conversions by printing data into the pipe and reading it back using a different format.

Data transfer through pipes is extremely efficient. Pipes can be used much like signals to coordinate processes with each other, but with these distinct advantages:

1. Longer (than 16 bits) messages
2. Queued messages
3. Determination of pending messages
4. Easy process-independent coordination (using named pipes)

Named and Unnamed Pipes

OS-9 supports both named and unnamed (anonymous) pipes. Unnamed pipes are used extensively by the Shell to construct program "pipelines." They may be freely used by user programs as well. Unnamed pipes may be opened only once. Independent processes may communicate through them only if the pipeline was constructed by a common parent to the processes. This is accomplished by making each process inherit the pipe path as one of its standard I/O paths.

The main difference between named and unnamed pipes is the same named pipe may be opened by several independent processes. This simplifies pipeline construction. In almost all other respects, named and unnamed pipes function identically. Specific differences are noted in the sections that follow.

Creating Pipes

The I$Create system call is used with the pipe file manager to create new **named** or **unnamed** pipe files.

Pipes may be created using the pathlist "/pipe" (for unnamed pipes, "pipe" is the name of the pipe device descriptor) or "/pipe/<name>" (<name> is the logical file name being created). If a pipe file with the same name already exists, an error (E$CEF) is returned. Unnamed pipes can not return this error.

All processes connected to a particular pipe share the same physical path descriptor. Consequently, the path is automatically set to "update" mode regardless of the mode specified at create. Access permissions may be specified, and are handled similar to RBF.

The size of the default fifo buffer associated with a pipe is specified in the pipe device descriptor. This may be overridden when creating a pipe by setting the initial file size bit of the mode byte and passing the desired "file size" in register d2. If no default or overriding size is specified, a small fifo buffer inside the path descriptor will be used. This buffer is currently 90 bytes.

Opening Pipes

When accessing unnamed pipes, I$Open works in the same way as I$Create. It opens a new anonymous pipe file. With named pipes, open searches for the specified name through a linked list of named pipes associated with a particular pipe device. If the pipe is found, the path number returned will refer to the same physical path that was allocated when the pipe was created. Internally, this works in a similar fashion to the I$Dup system call.

Opening an unnamed pipe is simple. A bit more complex is the method allowing another process to share the pipe. If you simply opened a new path to "/pipe" for the second process, the new path would be independent of the old one.

The only way for more than one process to share the same unnamed pipe is through the inheritance of the standard I/O paths through the F$Fork call. As an example, the outline given on the folowing page describes a method the shell might use to construct a pipeline for the command "dir -u ! qsort". Assume paths 0,1 are already open.

```
StdInp = I$Dup(0)            save the shell's standard input
StdOut = I$Dup(1)            save shell's standard output
         I$Close(1)          close standard output
         I$Open("/pipe")     open the pipe (as path 1)
         I$Fork("dir","-u") fork "dir" with pipe as standard output
         I$Close(0)          free path 0
         I$Dup(1)            copy the pipe to path 0
         I$Close(1)          make path available
         I$Dup(StdOut)       restore original standard out
         I$Fork("qsort")     fork qsort with pipe as standard input
         I$Close(0)          get rid of the pipe
         I$Dup(StdInp)       restore standard input
         I$Close (StdInp)    close temporary path
         I$Close (StdOut)    close temporary path
```

The main advantage of using named pipes is that several processes may communicate through the same named pipe, without having to inherit it from a common parent process. For example, the above steps can be approximated by "dir -u >/pipe/temp & qsort </pipe/temp".

NOTE: The OS-9 shell always constructs its pipelines using the unnamed "/pipe" descriptor.

Read/ReadLn

The I$Read and I$ReadLn system calls return the next bytes in the pipe FIFO buffer. If there is not enough data ready to satisfy the request, the process reading the pipe is put into a sleep state until more data becomes available.

The end-of-file is recognized when the number of processes waiting to read the pipe is equal to the number of users on the pipe. If any data is read before end-of-file is reached, an end-of-file error is not returned. The byte count returned however will be the number of bytes actually transferred.

NOTE: The Read and Write system calls are faster than ReadLn and WritLn because pipeman does not have to check for carriage returns and the loops moving data are tighter.

Write/WritLn

The I$Write and I$WritLn system calls work in almost the same way as I$Read and I$ReadLn. Instead of end-of-file being recognized, a pipe error (E$Write) occurs when data is written that can never be read (writing to a full pipe).

When named pipes are being used, pipeman never returns the E$Write error. If a named pipe becomes full before a process that receives data from the pipe has opened it, the process writing to the pipe is put to sleep until a process reads the pipe.

Close

When a pipe path is closed, its path count is decremented. If no paths are left open on an unnamed pipe, its memory is returned to the system. With named pipes, its memory is returned only if the pipe is empty. A non-empty pipe (with no open paths) is artificially kept open, waiting for another process to open and read from the pipe. This permits pipes to be used as a type of a temporary, self-destructing "RAM disk file".

Getstat/Setstat

Pipeman supports a wide range of status codes, to allow pipes to be inserted between processes where an RBF or SCF device would normally be used. For this reason, most RBF and SCF status codes are implemented to do something without returning an error. The actual function may differ slightly from the other file managers, but it is usually compatible.

GETSTAT STATUS CODES	
NAME	**FUNCTION**
SS_Opt	This reads the 128 byte option section of the path descriptor. It can be used to obtain: path type, data buffer size, name of pipe.
SS_Ready	This tests whether data is ready. It returns the number of bytes in the buffer.
SS_Size	This returns the size of the pipe buffer.
SS_EOF	This tests for end-of-file.
SS_FD	This returns a pseudo-file descriptor image.
Other codes are passed to the device driver.	

SETSTAT STATUS CODES	
NAME	**FUNCTION**
SS_Opt	This does nothing, but returns without error.
SS_Size	This sets the file size; resets the pipe buffer if the specified size is zero. Otherwise, it has no effect, but returns without error.

SETSTAT STATUS CODES	
NAME	**FUNCTION**
SS_FD	This does nothing, but returns without error.
SS_Attr	This changes the pipe file's attributes.
SS_SSig	This sends a signal when the data becomes available.
SS_Relea	This releases the device from the SS_SSig processing before data becomes available.
Other codes are passed to the device driver.	

The Makdir and Chgdir service requests are illegal service routines on pipes. They will return E$UnkSvc (unknown service request).

Pipe Directories

Opening an unnamed pipe in the "Dir" mode allows it to be opened for reading. In this case, pipeman allocates a pipe buffer and pre-initializes it to contain the names of all open named pipes on the specified device. Each name is null-padded to make a 32-byte record. This allows utilities, that normally read an RBF directory file sequentially, to work with pipes as well.

NOTE: Remember that pipeman is not a true directory device, so commands like "chd" or "makdir" do not work with /pipe.

The head of a linked list of named pipes is in the static storage of the pipe device driver (usually a "null" driver). If there are several pipe descriptors on a system, each having a different default pipe buffer size, the I/O system will notice that the same file manager, device driver, and port address (usually zero) are being used. It will not allocate new static storage for each pipe device and all named pipes will be on the same list.

For example, if two pipe descriptors exist, a directory of either device will reveal all the named pipes for both devices. If each pipe descriptor has a unique port address (0,1,...), the I/O system will allocate different static storage for each pipe device. This will produce more expected results.

PIPEMAN DEFINITIONS OF THE PATH DESCRIPTOR

The table shown below describes the option section (PD_OPT) of the path descriptor used by PIPEMAN.

NOTE: The term "offset" refers to the location of a module field, relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "sys.l" or "usr.l."

OFFSET		USAGE
$80	DV_DTP	device type
$81		Reserved
$82	PD_BufSz	Default pipe buffer size
$86	PD_IOBuf	Reserved I/O buffer
$E0	PD_Name	Pipe file name

Figure 25: Path Descriptor PD_OPT for PIPEMAN

NAME	UTILIZATION
DT_DTP	**Device type** 0 = SCF 1 = RBF 2 = PIPE 3 = SBF 4 = NET
PD_BufSz	**Default pipe buffer size** This contains the default size of the FIFO buffer used by the pipe. If no default size is specified and no size is specified when creating the pipe, PD_IOBuf will be used.
PD_IOBuf	**Reserved I/O buffer** This contains the small I/O buffer to be used by the pipe if no other buffer is specified.
PD_Name	**Pipe file name (if any)**

End Of Chapter 10

NETWORKING AND THE NETWORK FILE MANAGER

OVERVIEW OF NETWORKING

OS-9/NET is an optional extension for the OS-9 operating system which provides a powerful software-based network architecture. A key feature of the system is that the user interface is at the normal OS-9 file system level.

The networking system is based on a new Network File Manager (NFM) module which provides the same functions as the standard OS-9 disk file manager (RBF). Networking works with any I/O device (RBF, SCF, SBF, etc.). This allows files resident on remote systems to be accessed in an identical manner as if they were resident on a local disk.

Each system connected to the network ("node") has a logical name. Files or devices accessed are "opened" from a remote system by simply adding the logical system name to a standard OS-9 pathlist. For example, a command to list a file on a remote system (called "sys5") would look like:

```
list /net/sys5/d1/textfile
```

The network file managers on the local and remote nodes automatically and transparently convert all input/output requests to the necessary internal logical and physical network protocols. All disk-type file operation functions are supported, including creation and deletion of files and directories, changing working directories, and so on.

An additional level of file security positively controls network access to files on each node.

Network Hardware Compatibility

OS-9/NET was designed to be hardware-independent and compatible with all popular networking hardware such as OMNINET, ETHERNET, ARCNET and similar systems. All hardware dependencies are kept within a network device driver module. Therefore, adapting OS-9/NET to new networking hardware is similar to installing a new type of disk drive on an OS-9 system.

Requirements for Networking

To implement OS-9/NET, OS-9/68000 Version 1.2 (or later) is required.

Like all standard SCF/RBF devices, OS-9/NET requires both device drivers and device descriptors. In addition, if "multi-node" networking is being implemented, a "Node-ID Table" data module is required. This table describes the relationship between the physical station ID (node ID) and the corresponding station name (node name).

USER OVERVIEW

OS-9/NET allows access to all types of devices through the networking line. Each device is accessed by the standard OS-9 pathlist format:

```
/<network name>/<node name>/<device path>
```

For example:

```
/net/system1/h0/doc/manual
/net/system2/d2/backup/file
```

"/net" specifies that the device to be accessed is on the network. **"system1"** is the "node name" or station name. Like other OS-9 devices, the name of the network and the node itself is not "fixed". This allows a computer to be connected to more than one network. For example, you might name two types of network hardware connected to your system "/NET" and "/MODEM" and the nodes accessed by them system45, Hugo, Howard, Heckel and Jeckel. Following the node name is a full OS-9 pathlist beginning with the root directory (i.e. /h0/...).

CAVEAT: There exists one minor problem in using the "chd" built-in Shell command over network lines. A "chd pathlist" may only include one network station. For example:

```
chd /n0/heckle/h0
chd /n1/jeckle/h0/backups
```

The above commands will change your directory to /h0/backups on the remote station "jeckle". The following command will return no error itself, but any subsequent command (other than chd) will return an error:

```
chd /n0/heckle/n1/jeckle/h0/backups
```

To recover from this situation, execute a "single network chd" command. for example:

```
chd /n1/jeckle/h0/backups
```

Or

```
chd /h0
```

All other OS-9 commands will work across multi-network boundaries. For example, the following command will list the specified file:

```
list /n0/heckle/n1/jeckle/h0/backups/myfile
```

Multi-station Networking

Networks can be built using any combination of appropriate 68000 computers. Access to the different stations require the full network pathlist as shown previously. Because of OS-9/NET's hardware independence, most available networking hardware (OMNINET, ETHERNET, etc.) can be used.

NOTE: "Multi-station" networking requires the PD_HDTyp field in the device descriptor to be set to a non-zero value.

Point-to-point Networking

Point-to-point networking is the networking of two systems by way of RS-232 cables. Using this type of set-up, networking pathlists may be shortened:

```
/<network name>/<device path>
```

NOTE: Point-to-point networking requires the PD_HDTyp field in the device descriptor to be set to zero.

Multi-Network Systems

By adding more networking hardware (each with a unique device name), you may create a "multi-network" system. The Network File Manager and device drivers are re-entrant, as are all OS-9 File Managers and device drivers. This allows you the ability to add more networking ports simply by adding device descriptors corresponding to the added network hardware.

The Network Utilities

Two utilities are provided to allow easier use of OS-9/68000 Networking: Nmon and Ndir. Nmon is the system entry supervisor. It allows a station to enter or leave the network. Ndir is the network directory command. It displays the network stations currently running on the network. Both of these utilities are fully detailed in "Using Professional OS-9" and "Using Personal OS-9".

BROADCASTING OVERVIEW

Broadcasting is available through most commonly available networks (i.e. OMNINET, ARCNET, ETHERNET, etc.). It allows each station connected through the network to send out information to all currently running stations on the network. This is done when entering or leaving the network.

There are three types of broadcast messages used by OS-9/NET:

1. **RESET:** This message is broadcast when a station runs "nmon -u". This includes the station ID.

2. **DOWN:** This message is broadcast when a station runs "nmon -d". This also includes the station ID.

3. **RECONFIGURATION:** This message is broadcast from outlying stations upon receiving a RESET message. The newly reset station will receive a RECONFIGURATION message from each outlying station on the network.

When a station enters the network (runs "nmon -u"), it broadcasts the RESET message. Each station that receives this message will search its NET_Nodes module for the new station ID. If it is found, the receiving station first checks to see if there are any previously open paths from the new station. If so, the paths will be closed immediately. Any open paths from the receiving station to the new station will return a #173 error. Once this path protocol is finished, the receiving station will return a RECONFIGURATION message and the Nd_Actv field for the new station is set to TRUE (-1). Once Nd_Actv is set, Ndir will display this station.

If the new station is not found in the receiving stations NET_Nodes module, the RESET message is ignored. This prevents any station from entering the network that is not specified by NET_Nodes.

When a station leaves the network, it broadcasts the DOWN message. Each station that receives this message will search its NET_Nodes module for the station ID. If it is found, the receiving station again checks for open paths between the two stations. Any open paths between the stations will return a #173 error. Open paths from outlying stations will be closed on the system going down. Once this is finished, the Nd_Actv field is set to FALSE (0). Ndir will no longer display this station. If the new station is not found in the receiving stations NET_Nodes module, the message is ignored.

NOTE: Powering down without running "nmon -d" first or resetting the system without running "nmon -u" immediately is extemely discourteous and very annoying. If a system is reset or turned off without running "nmon -d", other systems on the network will still display the reset system as "active". All open paths from other stations will wait until the powered down or reset system is turned on again and/or "nmon -u" is run. If you know a system is down, but it is still listed as "active" by the ndir display, run "nmon -d" and "nmon -u" on your system to display an accurate network configuration.

NETWORK SECURITY AND THE "_USERS" MODULE

Commonly, small systems are only accessed by one user. This is the super-user. When small systems and large systems are networked together, the default security for OS-9 would allow the small system super-user access to all files on the large system. It would also shut out all non-super-users from the small system. This is not usually in the best interests of a network. Consequently, a security module can be made to allow and deny access to specified users.

OS-9/NET searches for the security module at startup. The security module must be named in the following manner:

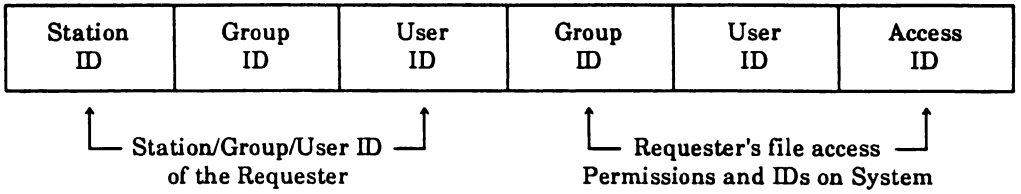
```
<network name> + "_users"
```

If the module is found, it will stay in memory during all network activity. All incoming requests are checked against entries in the security module for file access. The requests are consequently denied or allowed. If denied, the appropriate OS-9 error code is returned to the requester.

The security module is in standard OS-9 data module format:

Module Header (including offset to 1st entry)
. . .
1st entry
2nd entry
. . .
0 (terminator)
module CRC

Each entry in the security module is made up of six 16-bit fields. There is no limitation to the number of fields allowed. All other parts of the module (header, CRC, offset to 1st entry) are generated by the linker. Each field has the following format:



The Msg_Src and Msg_User fields of an incoming message are compared to the first three fields of each entry. If a match is found, the next three fields are used as the Requester's respective ID and access permissions. This is checked before allowing access to any device on the system receiving the incoming message. If the requester's station/group/user ID is not found in the table, access is denied.

NOTE: If the security module does not exist, all network file access will depend solely on group/user IDs.

Building A Security Module Entry

Requester Station ID: This may contain the requester's logical station ID or U_ANY. If U_ANY is specified, there are no restrictions set for the station ID. U_ANY is defined in net.l. Do not redefine it in the source code file.

Requester Group/User ID: This may contain the requester's group/user ID on their own system or U_ANY. If U_ANY is specified, there are no restrictions set for group/user ID. This is useful for default entries.

Assigned Group/User ID: This may contain specific group/user IDs that the requester will have on the system receiving the request. This field may also contain U_DFT or U_NODE. U_DFT assigns the requester's own group/user ID for system access. U_NODE assigns the requester's station ID as their group or user ID. U_NODE would be helpful, for example, in knowing the network origin of newly created files.

Assigned File Access Permissions: This may contain either "0" (indicating no access) or any combination of the following:

READ_
WRITE_
EXEC_

It is important to note that certain commands require these permissions, but are not commonly associated with them. For example, "chd" needs "WRITE__" permission, "load" and "chx" need "EXEC__" permission, etc.

Example Entries

The Security Module table is read from left to right and from top to bottom. Placement of entries is important. As the table is checked, the first entry that will agree with the message IDs will be used for assigning access permissions. Consequently, default access permissions should appear at the end of the table.

The following are example entries in a security module and descriptions of their effect on system access:

```
U_Start
        dc.w  U_ANY,0,0,U_NODE,U_NODE,0
        dc.w  U_ANY,0,U_ANY,77,U_NODE,Read_
        dc.w  U_ANY,5,U_ANY,U_DFT,U_NODE,Read_+Write_
        dc.w  U_ANY,U_ANY,U_ANY,U_DFT,U_NODE,Read_+Exec_
        dc.w  0
```

The first entry in the table above denies all super-users (0.0) access to the system. The second entry allows all "group 0" users (except 0.0) Read access with the group/user ID 77.U__NODE. The third entry allows all "group 5" users Read and Write access with the group/user ID 5.U__NODE. The fourth entry allows all other users (not previously specified) Read and Execution access with their own group ID and the user ID of U__NODE.

```
U_Start
        dc.w  69,0,0,U_NODE,U_NODE,0
        dc.w  U_ANY,0,U_ANY,0,U_NODE,Read_
        dc.w  0
```

The first entry denies access to any super-user from station "69". The second entry allows Read access to all users with group "0". It should be noted that if these two entries were reversed, user 0.0 from station "69" would be allowed Read access.

All security module tables end with "dc.w 0". It is possible to create an empty table by including only this line. This would allow no access to the station, but allow requests to originate from it.

ERROR TROUBLESHOOTING

The following reference shows the error number returned when accessing /NET/... and possible reasons for its occurrence.

ERROR	POSSIBLE CAUSES
#173	1. Nmon was killed on the station you were accessing. 2. The station was reset and a new "nmon -u" was exectuted on the station you were accessing.
#201	1. The "netserv" process was not found on the requested station. This may be caused by connecting a broadcasting system with a non-broadcasting system and one of the systems is reset while paths are open on the other. Another reason for not finding "netserv" is that it was killed for some reason. Check "procs" on the requested station.
#215	1. Check your pathlist for incorrect spellings or illegal characters. 2. Make sure your pathlist is neither incomplete nor incorrect: /net — incomplete /net/<station name> — incomplete /net/<station name>/<non-root directory> — incorrect /net/<non-root directory> — incorrect 3. The station name in the pathlist is not specified in the NET_Nodes module. Check "ndir"
#221	1. "NET_node" module is not found by "ndir". 2. The device on the requested system is not found or is incorrect.

ERROR	POSSIBLE CAUSES
#246	1. The device on the requested system is really not ready. Try again. 2. "nmon" is not running on your system. Check "procs" 3. The requested station is not on the network. Check "ndir /<net>" 4. The requested station's "nmon" or "eio" process was killed for some reason. Check "procs" on the requested station. Run "nmon -d" and then "nmon -u" on the requested station.
#250	1. The network device is busy. This frequently occurs with CSMA/CD type network configurations. This may also occur if the cable is disconnected, depending on controller characteristics. 2. "nmon -u" was invoked a second time.

NFM: THE NETWORK FILE MANAGER

The Network File Manager is a re-entrant subroutine package for I/O service requests across network lines. NFM can handle any number or type of networking systems simultaneously.

The internal functions of the Network File Manager are roughly divided into two parts. These co-operate with each other to allow the system to communicate through the network:

1. The "Request Sender" sends I/O requests to the network. It is the standard interface between the File Manager and the kernel/user. It receives a user I/O request, builds a message and sends it over the network. It then receives a service response from the node that received the message and reports the node's status and/or data to the user program. The Request Sender uses the device driver's "WRITE" entry point, regardless of type of I/O request.

   NOTE: This is only invoked when a user program executes an I/O system call.

2. The "Background Monitor/Server": This part acts as the network's incoming message monitor/server and response sender. It receives and parses incoming messages from other stations on the network. It then invokes a system process to execute the requested service on its associated system and returns the status/data to the requester over the network. If the incoming message is a response to a request, the message is passed to the Request Sender. The Monitor/Server uses the driver's "READ" entry point to allow the system to get responses from requests as well as receive requests from other systems.

   NOTE: This is invoked by the program "Nmon".

These functions are divided into several smaller functions, which are discussed in the next section.

TRACING A USER REQUEST THROUGH A NETWORK

To illustrate the interaction of NFM, the NFM device driver and the OS-9 kernel, it is useful to trace a user request through a simple network system. A graphic representation of the request is shown in Figure 26 and explained below.

1. User: the user program executes a service request to be given status and data (if available).

2. OS-9 kernel: Upon receiving the service request, the kernel dispatches it to the correct I/O handling modules (in some cases RBF or SCF is called). With OS-9, networking is handled exactly like any other device handling. The kernel does not care about the request destination. The Network File Manager is called if the request is outside the originating system. Consequently, the OS-9 network device is transparent to the system/user.

3. Network File Manager: The "Request Sender" receives specialized I/O requests from the kernel and builds a logical message in general OS-9/NET format. It handles the physical device controls and calls the device driver. It then waits for the message to complete transmission. Upon reception of the response from the driver, the "Request Sender" returns the status/data back to the kernel. In the background, the "Background Monitor/Server" receives two types of incoming logical messages from the Device Driver:

    1. Responses to previously sent requests are dispatched to waiting processes.

    2. Logical message/requests from outlying stations are invoked as separate service processes, which recursively calls the kernel, as necessary. After the process' completion, a message is sent back to the requester through the device driver.

    All system requests originating outside the system are processed by the Background Monitor/Server.

4. Device Driver: Upon receiving a logical message from the Network File Manager, the device driver sends it out onto the network line according to its physical characteristics. This could range from RS-232C, HDLC or EtherNet to large scale network communication lines.

    Usually a specialized hardware level protocol is used for communication as well as the physical method for accomplishing it. Data encryption and decryption are also done at this time.

    Incoming logical messages from outer stations are sent to the Network File Manager without touching their content. Interrupt driven or hardware level monitoring for the incoming messages must be done after the system is brought up.

    NOTE: The Device Driver never looks into the message contents; it merely passes them to the appropriate destination.

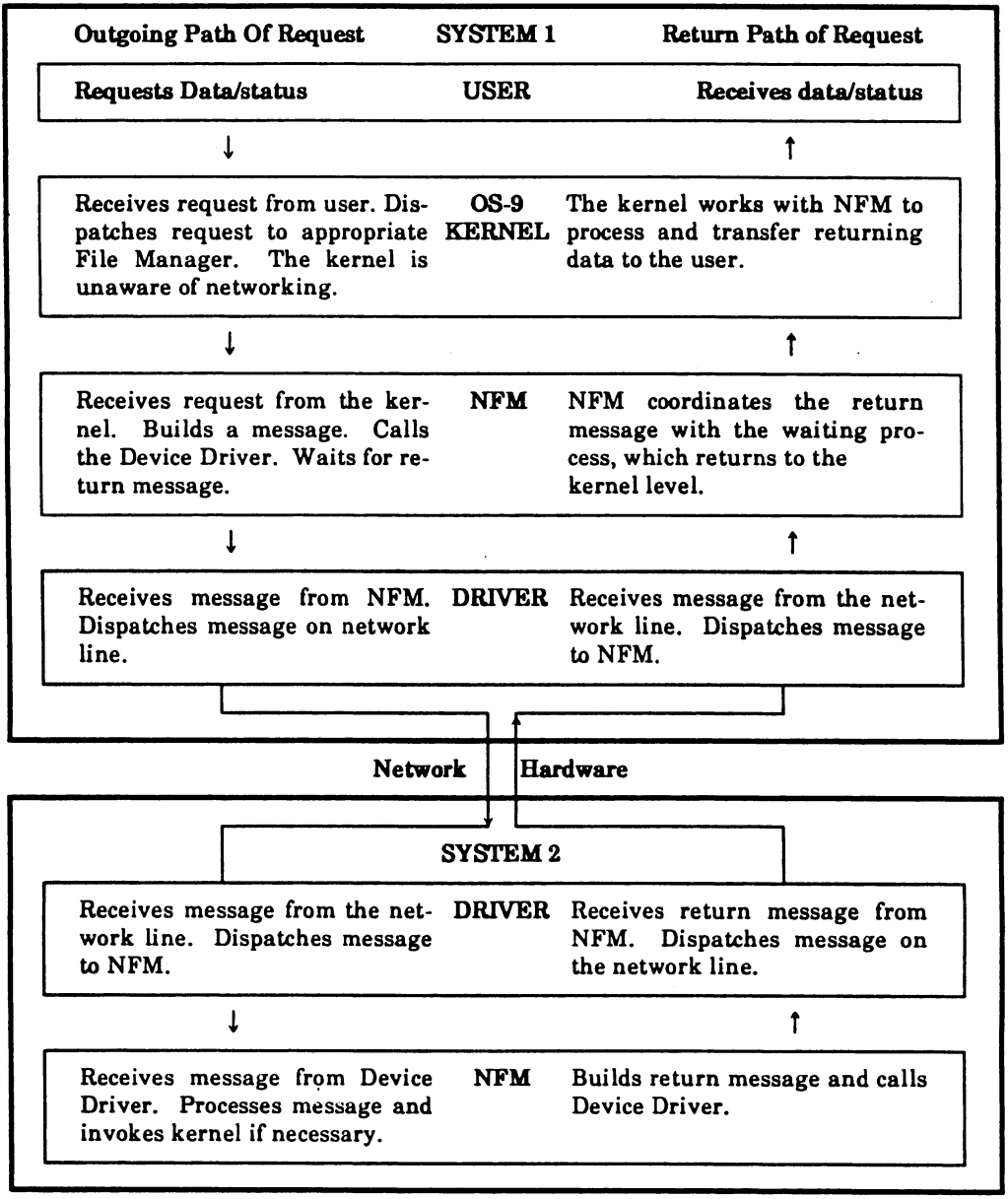


Figure 26: Tracing a user request through the network

THE OS-9/NET DEVICE DRIVER

An OS-9/NET device driver is like all other OS-9 device drivers. It is called by and works with the kernel and the Network File Manager. It conforms to the standard OS-9 memory module format (module type code: Drivr).

The most important function of the OS-9/NET driver is the handling of logical messages sent to or received from the Network File Manager. The driver uses the READ entry point to receive messages and transmits them using the WRITE entry point.

All messages built by the Network File Manager conform to a standard format. The following example is a message built when a I$Create system request is called.

Length	Name	Description
word	Msg__Dest	Destination Node ID
word	Msg__Src	Source Node ID: Message sender ID
word	Msg__CPU	Message Sender's OS-9 System Level
word	Msg__Code	Meaning of Message: Create
word	Msg__Path	NFM's Path# on the Requester's System
word	Msg__Pack	Requester's Node Packet Serial#
long	Msg__User	Requester's Group.User ID on the Requester's System
long	Msg__NUsr	Security Information passed between NFMs
word	Msg__Len	Message Length: Message begins at the next field
word	RqCr__Mod	I$Create System Call Parameter
word	RqCr__Atr	I$Create System Call Parameter
word	RqCr__Isz	I$Create System Call Parameter
10 bytes	RqCr__Dft	Requester's Login-Directory Information
variable	RqCr__Ptl	I$Create Parameter String, terminated by NULL

The first nine fields are standard fields used in all messages. The remaining fields are system call dependent (in the above example: I$Create). The driver examines only two of these fields:

Msg_Dest Gives the driver the destination of the message. This is used in the Write entry.

Msg_Len Gives the driver the length of the message. It is also used in the Read and Write entry.

All other fields are strictly used by the File Manager or a specific OS-9 device. This cluster of information is passed between OS-9 operating systems, not drivers. The driver is totally unaware of the contents of the message. It does not change or examine any part of the message other than the fields specified above. The driver merely sends the message of the length specified by Msg_Len (plus header) to the node specified by Msg_Dest. A hardware driver usually adds its individual header and trailer (i.e., a CRC check for physical error detection/correction).

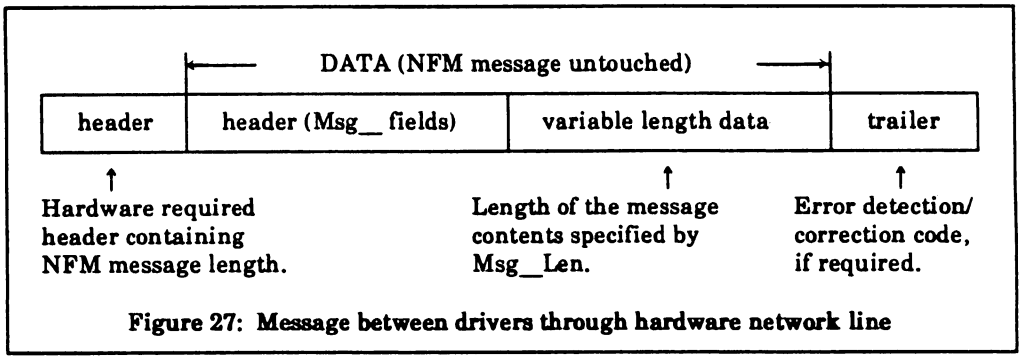


Figure 27: Message between drivers through hardware network line

The driver may have its own protocol and the hardware may also have its own protocol. Because hardware characteristics vary from system to system, the Network File Manager does not concern itself about the independent hardware protocol. This is the responsibility of the device driver. The driver is solely responsible for handling the hardware, the network line, adding its individual header (and if required trailer) and if necessary exchanging some sequential hardware handshaking protocol.

NFM DEVICE DESCRIPTOR MODULES

This section describes the definitions of the initialization table contained in device descriptor modules for the Network File Manager. These values are copied into the corresponding option field of the path descriptor at initialization (when Nmon is invoked or the device is "iniz"-ed). This table immediately follows the standard device descriptor module header fields. The size of the table is defined in the M$Opt field. A graphic representation of the table is supplied below:

NOTE: The term "offset" refers to the location of a module field relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "net.l".

Offset	Name	Usage
$48	PD_DTP	Device Type
$49	PD_HdTyp	Network Hardware Type
$4A	PD_BC	Hardware Broadcast Availability
$4C	PD_Dest	Default Destination ID
$4E	PD_Src	Node/Station ID
$50	PD_Broad	Harware Broadcast Code
$52	PD_RtyF	# of Retries for NFM
$54	PD_RtyD	# of Retries for Device Driver
$56	PD_Mul	Multiplier for Retries
$58	PD_HWB	Hardware Message Buffer Address
$5C	PD_NPAR	Parity for RS232 Networking
$5D	PD_NBAU	Baud Rate for RS232 Networking
$5E	PD_EXTR	User-defined fields begin here

Figure 28: Device Descriptor Initialization Table

NAME	UTILIZATION
PD_DTP	**Device Class** (0=SCF 1=RBF 2=PIPE 3=SBF 4=NET)

PD_HdTyp	**Network Hardware Type**
	If set to a non-zero value, this hardware is connected to a multi-station network line. The Network File Manager then expects a full network pathlist:
	`/<network name>/<station name>/<full OS-9 pathlist>`
	If set equal to zero, "point-to-point" networking is assumed. The Network File Manager then expects a simple network pathlist:
	`/<network name>/full OS-9 pathlist>`

PD_BC	**Hardware Broadcast Availability** If this field is set to -1, hardware broadcasting is available.
PD_Dest	**Default Destination Station/Node ID** This field is used by the Network File Manager as the default destination station/node ID. It is copied to the path descriptor at initialization. When PD_HdTyp is non-zero, the Network File Manager updates the corresponding field in the path descriptor for the destination of each network message, while this field (in the device descriptor) remains unchanged. This field is not used when point to point networking is specified.
PD_Src	**Source Station/Node ID** This is used by the Network File Manager as the source ID for all outgoing messages. If this field is set to -1 ($FFFF), NFM assumes the device driver for this device will set the actual/working node ID in V_myID of the device driver's static storage. NFM then updates this field in the path descriptor with the value from V_myID, while this field remains -1. This insures only one source code for the device descriptor. The device driver determines the node ID by checking the actual hardware.
PD_Broad	**Hardware Broadcast Code** This is solely used by the device driver.

NAME	UTILIZATION

PD_RtyF **Number of Re-Tries For Network File Manager**
Some networking hardware return errors if a device is busy. When the Network File Manager detects an E$DevBsy error (passed by the driver), it will try to send the message over again. This field's value, when multiplied by the value of PD_Mul, specifies how many times it will try to send a message before returning an error.

PD_RtyD **Number of Re-Tries For Device Driver**
Some networking hardware return errors if a device is busy. When the device driver detects an E$DevBsy error, it will try to send the same message over again. This value, when multiplied by the value of PD_Mul, specifies how many times it will try to send a message before returning an error. The value "0" specifies only one try.

PD_Mul **Re-Try Multiplier**
See PD_RtyF and PD_RtyD.

PD_HWB **Harware message buffer address**
Some hardware use a physical buffer for storage that is different from its port address. If so, the address should be placed here. This field is only used by the device driver.

PD_NPAR **Parity code for RS-232 networking**
Bits 0 and 1 indicate the parity as follows:

```
0 = no parity
1 = odd parity
2 = even parity
```

Bits 2 and 3 indicate the number of bits per character as follows:

```
3 = 5 bits per character
2 = 6 bits per character
1 = 7 bits per character
0 = 8 bits per character
```

Bits 4 and 5 indicate the number of stop bits as follows:

```
0 = 1 stop bit
1 = 1 1/2 stop bits
2 = 2 stop bits
```

Bits 6 and 7 are reserved.

We recommend using the same value as PD_PAR in the SCF device descriptor.

NAME	UTILIZATION

PD_NBAU **Baud Rate for RS232 networking**

The Baud rate is set as follows:

0 = 50 baud	6 = 600 baud	C = 4800 baud
1 = 75 baud	7 = 1200 baud	D = 7200 baud
2 = 110 baud	8 = 1800 baud	E = 9600 baud
3 = 134.5 baud	9 = 2000 baud	F = 192000 baud
4 = 150 baud	A = 2400 baud	FF = External
5 = 300 baud	B = 3600 baud	

We recommend using the same value as PD_BAU in the SCF device descriptor.

PD_EXTR **User-defined fields**

If specific hardware requires descriptor fields other than those above, it is recommended to offset the fields at "PD_EXTR + xxx". For example, certain hardware uses a different port for its IRQ status. This could be described at "PD_EXTR + IRQPort". The device driver's code in "Init" would be:

```
movel  M$DTyp + PD_EXTR + IRQPort - PD_OPT(a1),V_IPORT(a2)
```

NFM PATH DESCRIPTORS

The reserved section (PD_OPT) of the path descriptor used by NFM is copied directly from the initialization table of the device descriptor. The following table is provided to show the offsets used in the path descriptor. For a full explanation of the path descriptor fields, refer to the previous pages.

NOTE: The term "offset" refers to the location of a module field relative to the starting address of the module. Module offsets are resolved in assembly code by using the names shown here and linking the module with the relocatable library: "net.l".

Offset		Usage
$80	PD_DTP	Device Type
$81	PD_HdTyp	Network Hardware Type
$82	PD_BC	Hardware Broadcast Availability
$84	PD_Dest	Default Destination ID
$86	PD_Src	Node/Station ID
$88	PD_Broad	Harware Broadcast Code
$8A	PD_RtyF	# of Retries for NFM
$8C	PD_RtyD	# of Retries for Device Driver
$8E	PD_Mul	Multiplier for Retries
$90	PD_HWB	Hardware Message Buffer Address
$94	PD_NPAR	Parity for RS232 Networking
$95	PD_NBAU	Baud Rate for RS232 Networking
$96	PD_EXTR	User-defined fields begin here

Figure 29: Path Descriptor Option Table

NODE NAME DATA MODULE

The Network Node Name Data Module specifies the relationship between a node/station name and its corresponding node/station ID. This module is only used when the Network Device Descriptor field, PD_HdTyp is non-zero, specifying multi-station networking. The name of this module must be the name of the network device with a "_nodes" suffix. For example:

Device	Module Name
/net	net_nodes
/ether	ether_nodes
/omni	omni_nodes

This module is automatically loaded/linked at the time "Nmon" is first invoked. If PD_HdTyp is non-zero and this module is not in the current execution directory, the system will not start, causing an "Nmon" error. If PD_HdTyp is set to zero, the system never uses this module, even when the hardware requires an actual destination ID. In this case, the driver is responsible to set the ID and establish linkage.

For a driver to be able to use the ID specified in Msg_Dest (and Pd_Dest), it must be described in this module. For each node/station to be used in the network, there is an entry for both the node ID and its name string. Each node ID entry has six fields:

Size	Use
Long	Offset to corresponding node name string
Word	Node ID
Byte	Nd_Actv Flag. If PD_BC = TRUE (hardware broadcasting is available), then set to 0 if the node is silent and set to -1 if the node is active. If PD_BC is FALSE, this field is unused.
Byte	Reserved
Long	Date the system is brought up on the network
Long	Time the system is brought up on the network

In the following example module, 2 node/stations are described:

Module Header
Offset to Entry 0
Offset to Name String for Name0 (Long) Node ID for Name0 (Word) Active Flag for Name0 (Byte) Date the station entered network (Long) Time the station entered network (Long)
Offset to Name String for Name1 (Long) Node ID for Name1 (Word) Active Flag for Name1 (Byte) Date the station entered network (Long) Time the station entered network (Long)
0 (terminator) (Long)
Node Name String for Name0
Node Name String for Name1
Module Name String
Module CRC

Figure 30: Example Node Name Module

NOTE: The network Node Name module contains flag fields for each entry that are changed each time the network configuration is changed. Consequently, this module is not ROMable.

NFM DEVICE DRIVER STATIC STORAGE AND SUBROUTINES

NFM device driver modules contain a package of subroutines that handle the transfer of logical messages to and from a specific hardware controller. Because these modules are re-entrant, one copy of the module can simultaneously run several identical controllers.

An individual static storage area is allocated for each copy of the device driver. The size of this storage is given in the device driver module header (M$MEM). At start-up the device static storage will be cleared, with the exception of V_PORT. V_PORT will contain the device port address. Do not initialize any portion of the static storage with the exception of V_myID. The static storage is used as follows:

Offset		Usage
$00	V_PORT	Device Base Address
$04	V_LPRC	Last Active Process ID
$06	V_BUSY	Active Process ID
$08	V_WAKE	Process ID To Awaken
$0A	V_Paths	Linked List Of Open Paths
$0E		Reserved
$2E	V_Nmon	Nmon Process ID
$30	V_EIO	Parent Process ID for Background Services
$32	V_EIOmsg	Communication Area for Nmon and EIO

Offset		Usage
$36	V_NPPT	Pointer to NFM Path Table
$3A	V_RPT	Pointer to Table of "Request" Paths
$3E	V_PTT	Pointer to Table of "Service" Paths
$42	V_SActv	Active Remote Station flag (Point-to-Point networking only)
$43		Reserved
$44	V_NodMdl	Node Name Module Address
$48	V_UsrMdl	Online Service User Table
$4C	V_Pack	Packet Serial Number
$4E	V_myID	Node ID

Figure 31: NFM Device Driver Static Storage

As with all device drivers, the execution offset address in the module header points to a branch table corresponding to device driver subroutines. The entry table for NFM device drivers is as follows:

```
ENTRY   dc.w  INIT      initialize device
        dc.w  READ      read message from network
        dc.w  WRITE     send message to specific station
        dc.w  GETSTA    get device status
        dc.w  SETSTA    set device status
        dc.w  TERM      terminate device
```

Each subroutine should exit with the condition code register Carry bit cleared if no error occurred. Otherwise, the Carry bit should be set and an appropriate error code returned in the least significant word of register d1.

The TRAP entry should be defined as the offset to exception handling code or zero if no handler is available. This entry point currently not used by the kernal. However, in future revisions this will be accessed. 11-23

NAME: INIT

INPUT:
```
(a1) = address of device descriptor module
(a2) = address of device static storage
(a6) = system global data pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc=Carry bit set
d1.w=Error code
```

FUNCTION: INITIALIZE DEVICE AND ITS STATIC STORAGE

The INIT routine must:

1. Initialize the device static storage. This consists of the following:

   A. Setting the value of V_myID, if device descriptor's station ID is set to "-1" ($FFFF). This value is used by NFM.

   B. Setting the device driver's portion of static storage initial values.

2. Place the driver IRQ service routine on the IRQ polling list by using the OS-9 F$IRQ service request.

3. Initialize the device control registers.

4. Enable interrupts for message-input from the network.

NOTE: Prior to being called, the device static storage will be cleared (set to zero) except for V_PORT which will contain the device port address. Do not initialize any portion of static storage used by NFM except for V_myID.

NAME: READ

INPUT: (a1)=address of path descriptor "/NET@"
(a2)=address of device static storage
(a3)=address of 256-byte free buffer
(a4)=address of Nmon's process descriptor
(a6)=system global data pointer

OUTPUT: (a3)=pointer to input message, message on 256-byte buffer

ERROR OUTPUT: cc=Carry bit set
d1.w=Error code

FUNCTION: READ A MESSAGE FROM THE NETWORK

The READ routine inputs one message from the network, edits it (if necessary) into a NFM logical message and passes it to NFM. A typical sequence is as follows:

1. Assume input-interrupt has been enabled in INIT or previous READ.

2. If any message is available in the hardware buffer, pass it to NFM. If not, sleep until an IRQ routine wakes it up.

3. If necessary, edit the input message into the NFM logical format. The driver should strip out its private header, trailer or error correction code from the physical buffer, so that NFM will receive only the logical portion of it. If required, the driver may have to communicate with the message sender by itself, by the physical protocol.

4. Enable input interrupt again. Pass the message to NFM.

NOTES: If READ has detected a physical error and if it is impossible to restore the sender's logical message, READ does not necessarily need to return to the caller. In other words, READ is not expected to return errors. If a return to the caller is specified (Nmon in system state), set the Carry bit in the condition code register and the error code in d1.w and (a3). In any case, READ must make the message sender recognize that the message is denied.

The (a3) buffer is necessary even when the driver returns an error. On errors, the unused memory space is returned to the caller. Returning the input (a3) may be safest. (a3) on output is not required to be the same as the input. The given message at any address is processed by the system. If the driver keeps track of private buffers, make sure that:

a) the output (a3) points to normal memory space for OS-9.

b) the driver returns the buffer memory to the system when no longer needed.

READ is solely called by "nmon" in system state. "Nmon" always calls READ as long as it runs. Other processes will get CPU time only while "nmon" is asleep, that is, while READ is asleep waiting for an input message.

NOTE: There is no CPU traffic control between READ and WRITE. WRITE is called while READ is sleeping. If the hardware allows READ interrupts during WRITE's critical operations, systems may crash depending on the controllers characteristics. It is the driver's responsibility to handle this situation.

NAME: WRITE

INPUT: (a1) = address of path descriptor
(a2) = address of device static storage
(a3) = address of NFM logical message (Max 256-byte)
(a4) = address of current process' descriptor
(a6) = system global data pointer

OUTPUT: None

ERROR OUTPUT: cc=Carry bit set
dl.w=Error code

FUNCTION: SEND NFM MESSAGE TO SPECIFIED STATION

A typical sequence for intelligent-transferring hardware is as follows:

1. Copy the NFM message, adding the driver's private header, trailer and/or error correction code to the hardware buffer.

   The sum of Msg$Size + Msg_Len determines the total NFM message length. Usually, a driver handles it as a string of data. Presently, the message length is guaranteed to be less than or equal to 256 bytes.

   The message destination is given at Msg_Dest(WORD). If the destination contains a value of 0, the WRITE call is broadcasting. NOTE: some types of hardware return a different status after a broadcast.

2. If the device is not ready, enable its IRQ and sleep until the IRQ awakens it.

3. Send the message to the destination.

4. If necessary, the driver should communicate with the message receiver, by physical protocol. Some errors can be detected by this operation. The driver is responsible for detecting and returning an error if the message was not successfully transmitted. WRITE should not return an error if the message has been sent to the destination. An error should be returned only if you are able to cancel the message or abort "sending" of the message in progress. NFM assumes that WRITE returns an error only when the "sending" of the message failed.

5. Return to NFM with its status code.

NOTES: WRITE calls are queued by the system, consequently the driver should not concern itself about device-monopolization control by WRITE.

If a READ IRQ causes an interrupt during a WRITE operation, it is the driver's responsibility to control it.

If the returning error code is E$DevBsy, WRITE is "retried" several times (according to the descriptor). Retrying WRITE is not always necessary.

Calling WRITE with a broadcast message does not expect error status to be returned. However, returning from a "broadcast" indicates that all active stations have recognized the broadcast (unless their hardware is not working). If necessary, the driver should broadcast the message several times until all known nodes receive it.

NAME: GETSTAT/SETSTAT

INPUT: d0.w=Status code
(a1)=address of path descriptor
(a2)=address of device static storage
(a6)=system global data pointer

OUTPUT: Depends on the function code.

ERROR OUTPUT: cc=Carry bit set
d1.w=Error code

FUNCTION: GET/SET DEVICE STATUS

No function is currently defined for the driver presently. Both GetStat and SetStat return:

```
move.w  #E$UnkSvc,d1
ori     #carry,ccr
rts
```

NAME: TERMINATE

INPUT: (a1)=address of path descriptor
(a2)=address of device static storage
(a6)=system global data pointer

OUTPUT: None

ERROR OUTPUT: None

FUNCTION: TERMINATE DEVICE

When "nmon/NET -d" is executed or "nmon" is killed, the system sends out a broadcast DOWN message through the driver's WRITE entry. At the same time, the system closes (or fails with an error) the following paths:

1. All paths that processes on your station are using.

2. All paths that are requested by outlying stations.

TERMINATE is called after all this is done. It will terminate any command sequence, stop any interrupt and if possible disconnect the station from the network.

NAME: IRQ Service routine

INPUT: (a2)=address of device static storage
(a3)=port address
(a6)=system global data pointer

FUNCTION: SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device driver module branch table and is not called directly by NFM, it is a key routine in interrupt driven device drivers. Its function is to:

1. Poll the device. If the interrupt is not caused by this device, the carry bit must be returned set with an RTS instruction as quickly as possible.

2. Service device interrupts.

3. Send wake up signals to the waiting processes in READ/WRITE if they exist, when I/O is complete.

   NOTE: There may be two processes, one in WRITE and one in READ, each waiting for wake up signals from an IRQ service routine. This is a major difference from RBF or SCF device drivers.

4. When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.

NOTES

THE DEFS FILES

USING THE DEFS FILES

The DEFS directory on the system disk contains relocatable object modules that contain common system wide symbolic definitions. Symbolic names occur in almost all assembly language routines. For example, "I$Read" and "I$Write" are examples of symbolic names.

Symbolic names can be defined using EQU or SET instructions that are used in the program, but it is generally more convenient to use the system definition libraries. Using the libraries also minimizes the chance of error and improves the maintainability of programs.

The actual codes that the names represent are referenced by linking the program with "sys.l" or "usr.l" as a library file. In the event that a future release of OS-9 changes some definitions, you only need to relink your original program with the new library.

Sys.l contains definitions for all names accessible to routines that are part of the operating system. Many definitions are inaccessible or of no use to user programs. The library "usr.l" is a subset of "sys.l" but contains only the definitions likely to be needed in user programs. In addition to definitions of system calls, error codes, and memory module formats, the libraries also have convenience definitions for ASCII characters, register names, etc.

The source files that make up the libraries are included for documentation and educational purposes. We recommend that you do not modify the source files. If you would like to add definitions to one of the libraries, create a new source file and merge the relocatable output with the library. You can also create your own libraries.

There are individual DEFS files that generally correspond to different parts of the system. For example, definitions used in the RBF device drivers are contained in a file called "rbfstat.a".

The libraries contain the following files:

```
Usr.l:  funcs.a      Sys.l:  All files in Usr.l plus the following:
        process.a            sysio.a       ioglob.a
        module.a             sysglob.a     iodev.a
        io.a                 SCFstat.a     SCFdev.a
                             RBFstat.a     RBFdev.a
                             DRVstat.a
```

NOTE: The DEFS have changed between Version 1.2 and Version 2.0 of the operating system. All system state object modules and some user programs will need to be relinked or re-made.

You are encouraged to examine the content of each source file of interest to you. They can be a great learning tool for understanding OS-9.

The "oskdefs.d" file is different than the other DEFS files. The definitions it contains are simple expressions that are usually used in assembly statements, which in turn do not allow external references such as PSECT by the OS-9 linker. Consequently, "oskdefs.d" must be included in files by a "USE" statement in your assembler source file. The statements generally look like:

```
ifpl
use /dd/defs/oskdefs.d
endc
```

End of Chapter 12

SECTION 3 - TRAP HANDLERS AND EVENTS

- User Trap Handler Modules
- The Math Module
- Events and Semaphores

USER TRAP HANDLER MODULES

TRAP HANDLERS

The 68000 family of microprocessors has sixteen software trap exception vectors. The first (trap 0) is reserved for making OS-9 system calls. The remaining fifteen may be used as service requests to (user defined) "user trap handlers".

Microware provides standard trap handlers for I/O conversions in the "C" language, floating point math and trigonometric functions. The following traps are reserved:

trap 13: CIO is automatically called for any "C" program

trap 15: math is called for floating point math, extended integer math and/or type conversion. It is also used for programs using transcendental and/or extended mathematical functions.

For further information about math, refer to Chapter 14.

A user trap handler is an OS-9 module that usually contains a set of related subroutines. Any user program may dynamically link to the user trap handler and call it at execution time. It should be noted that while trap handlers reduce the size of the execution program, they do not do anything that could not be done by linking the program with appropriate library routines at compilation time. In fact, programs that call trap handlers are executed slightly slower than linked programs that perform the same function.

Trap handlers must be written in a language that compiles to machine code (such as assembly language or "C"). They should be suitably generic to be used by a number of programs.

Trap handlers are similar to normal OS-9 program modules with one exception. They have three execution entry points: a trap execution entry point, trap initialization entry point and trap termination entry point.

Trap handler modules have the module type of "TrapLib" and the module language of "Objct".

The trap module routines are usually executed as though they were called with a "jsr" instruction (except for minor stack differences). Any system calls or other operations that could be performed by the calling module are usable in the trap module.

It is possible to write a trap handler module that runs in system state. This is rarely advisable, but sometimes necessary. For a discussion of the uses of system state, refer to Chapter 3.

INSTALLING AND EXECUTING TRAP HANDLERS

A user program installs a trap handler by executing the F$TLink system request. When this is done, the OS-9 kernel links to the trap module, allocates and initializes its static storage (if any) and executes the trap module's initialization routine.

Typically the initialization routine has very little to do. It could be used to open files, link to additional trap or data modules, or perform other startup activities. It will be called only once (per trap handler) in any given program.

A trap module that will be used by a program is usually installed as part of the program's initialization code. At that time, the particular trap number (1-15) is specified that will refer to the trap module. The program will invoke functions in the trap module by using the 68000 "trap" instruction corresponding to the trap number specified. This is followed by a function word that is passed to the trap handler itself. The arrangement is very similar to making a normal OS-9 system call.

The OS-9 relocatable macro assembler has special mnemonics to make trap calls more apparent. These are "OS9" for trap 0, and "tcall" for the other user traps. They work like built-in macros, generating code as illustrated below:

OS-9 CALL:	EQUIVALENT ASSEMBLY LANGUAGE SYNTAX:
`OS9 F$TLink`	`trap 0` `dc.w F$TLink`
`tcall T$Math,T$DMul`	`trap T$Math` `dc.w T$DMul`

From user programs, it is possible to delay installing a trap module until the first time it is actually needed. If a trap module has not been installed for a particular trap when the first "tcall" is made, OS-9 checks the program's exception entry offset ("M$Excpt" in the module header). If it is zero, the program is aborted. Otherwise, OS-9 passes control to the exception routine. At this point, the trap handler can be installed, and the first tcall reissued. The second example in this chapter shows how this is accomplished.

TWO EXAMPLES: CALLING A TRAP HANDLER

The actual details of building and using a trap handler are perhaps best explained by means of a simple complete example.

EXAMPLE ONE: The following program ("TrapTst") uses trap vector 5. It first installs the trap handler and then calls it twice.

```
                nam     TrapTst
                ttl     example one - link and call trap handler
                use     defsfile
Edition         equ     1
Typ_Lang        equ     (Prgrm<<8)+Objct
Attr_Rev        equ     (ReEnt<<8)+0
                psect   traptst,Typ_Lang,Attr_Rev,Edition,0,Test

TrapNum         equ     5                   trap number to use
TrapName        dc.b    "trap",0            name of trap handler

Test            moveq   #TrapNum,d0         trap number to assign
                moveq   #0,d1               no optional memory override
                lea     TrapName(pc),a0     ptr to name of trap handler
                os9     F$TLink             install trap handler
                bcs.s   Test99              abort if error
                tcall   TrapNum,0           call trap function #0
                bcs.s   Test99              abort if error
                tcall   TrapNum,1           call trap function #1
                bcs.s   Test99              abort if error
                moveq   #0,d1               exit without error
Test99          os9     F$Exit              exit
                ends
```

EXAMPLE TWO: The following example shows how the preceding program could be modified to install the trap handler in an exception routine when the first "tcall" is executed. This might be done for a trap handler that may not be used at all by a program, depending on circumstances.

It accomplishes the exact same process as EXAMPLE ONE, but does not initialize the trap handler before using it. Instead, it provides a "LinkTrap" subroutine to automatically install the trap handler when it is first used. See TRACE OF EXAMPLE TWO later in this chapter.

```
                nam     TrapTst
                ttl     example two - call trap handler
                use     defsfile
Edition         equ     1
Typ_Lang        equ     (Prgrm<<8)+Objct
Attr_Rev        equ     (ReEnt<<8)+0
                psect   traptst,Typ_Lang,Attr_Rev,Edition,0,Test,LinkTrap
```

EXAMPLE TWO (continued)

```
TrapNum   equ     5                   trap number to use
TrapName  dc.b    "trap",0            name of trap handler

*******************************
* Main program entry point

Test:   tcall   TrapNum,0           call trap function #0
        bcs.s   Test99              abort if error
        tcall   TrapNum,1           call trap function #1
        bcs.s   Test99              abort if error
        moveq   #0,d1               exit without error
Test99  os9     F$Exit              exit

*******************************
* Subroutine LinkTrap
* Installs trap handler, and then executes first trap call.
* Note: error checking minimized to keep example simple.

* Passed: d0-d7=caller's registers
*   a0-a5=caller's registers
*   (a6)=data base address
*   (a7)=trap stack (see below)
* Returns: trap installed, backs up PC to execute "tcall" instr

* The "trap stack" looks like this:
*          .-----------------------.
*       +8 |   caller's return PC  |
*          >-----------------------<
*       +6 |  vector #             |
*          >-----------------------<
*       +4 | func code             |
*          >-----------------------<
*          |  caller's a6 register |
*    (a7)->'-----------------------'

LinkTrap: addq.l  #8,a7               discard excess stack info
          movem.l d0-d1/a0-a2,-(a7)   save registers
          moveq   #TrapNum,d0         trap number to assign
          moveq   #0,d1               no optional memory override
          lea     TrapName(pc),a0     ptr to name of trap handler
          os9     F$TLink             install trap handler
          bcs.s   Test99              abort if error
          movem.l (a7)+,d0-d1/a0-a2   retrieve registers
          subq.l  #4,(a7)             back up to tcall instruction
          rts                         return to tcall instruction
          ends
```

AN EXAMPLE TRAP HANDLER

The listing below shows the trap handler itself. It has been kept artificially simple to avoid confusion. Most trap handlers have several functions, and generally begin with a dispatch routine based on the function code.

```
            nam     Trap Handler
            ttl     Example trap handling module
            use     defsfile
Type        set     (TrapLib<<8)+Objct
Revs        set     ReEnt<<8
            psect   traphand,Type,Revs,0,0,TrapEnt
            dc.l    TrapInit            initialization entry point
            dc.l    TrapTerm            termination entry point

********************************
* Subroutine TrapInit
*   Trap handler initialization entry point

* Passed: d0.w=User Trap number (1-15)
*   d1.l=(optional) additional static storage allocated
*   d2-d7=caller's registers (parameters required by handler)
*   (a0)=trap handler module name ptr (updated)
*   (a1)=trap handler execution entry point
*   (a2)=trap module ptr
*   a3-a5=caller's registers (parameters required by handler)
*   (a6)=data base address

* Returns: (a0)=updated trap handler name ptr
*   (a1)=trap handler execution entry point
*   (a2)=trap module ptr
*   cc=carry set, d1.w=error code if error
*   Other values returned are dependent on the trap handler

* The stack looks like this:
*          .------------------------.
*       +8 |    caller's return PC  |
*          >------------------------<
*       +4 |    0000     |    0000  |
*          >-------------|----------<
*          |   caller's a6 register |
*    (a7)->'------------------------'

TrapInit  movem.l (a7)+,a6-a7         no initialization needed;
          rts                         simply return

********************************
* Subroutine TrapEnt
* User Trap entry point
```

EXAMPLE TRAP HANDLER (continued)

```
* Passed: d0-d7=caller's registers
*   a0-a5=caller's registers
*   (a6)=data base address

* Returns: cc=carry set, d1.w=error code if error
*   Other values returned are dependent on the trap handler

* The stack looks like this:
*          .-----------------------.
*       +8 |   caller's return PC  |
*          >-----------------------<
*       +6 |  vector #             |
*          >-----------------------<
*       +4 | func code             |
*          >-----------------------<
*          |  caller's a6 register |
*    (a7)->'-----------------------'

          org     0                   stack offset definitions
S.d0      do.l    1                   caller's d0 reg
S.d1      do.l    1                   caller's d1 reg
S.a0      do.l    1                   caller's a0 reg
S.a6      do.l    1                   caller's a6 reg
S.func    do.w    1                   trap function code
S.vect    do.w    1                   vector number
S.pc      do.l    1                   return pc

TrapEnt:  movem.l d0-d1/a0,-(a7)      save registers
          move.w  S.func(a7),d0       get function code
          cmp.w   #1,d0               is function in range?
          bhi.s   FuncErr             abort if not
          beq.s   Trap10              branch if function code #1
          lea     String1(pc),a0      get first string ptr
          bra.s   Trap20              continue

Trap10    lea     String2(pc),a0      get second string ptr
Trap20    moveq   #1,d0               standard output path
          moveq   #80,d1              maximum bytes to write
          os9     I$WritLn            output the string
          bcs.s   Abort               abort if error
Trap90    movem.l (a7)+,d0-d1/a0/a6-a7 restore regs
          rts                         return

FuncErr   move.w  #1<<8+99,d2         abort (return error 001:099)
Abort     move.w  d1,S.d1+2(a7)       return error code in d1.w
          ori     #Carry,ccr          return carry set
          bra.s   Trap90              exit

String1   dc.b    "Microware Systems Corporation",C$CR,0
String2   dc.b    "    Quality keeps us #1",C$CR,0
```

EXAMPLE TRAP HANDLER (continued)

```
*******************************
* Subroutine TrapTerm
* Terminate trap handler servicing.
* As of this release (OS-9 V1.2) the trap termination entry
* point is never called by the OS-9 kernel.  Documentation
* details will be available when a working implementation
* exists.

TrapTerm  move.w  #1<<8+199,d1      never called, if it gets here..
          os9     F$Exit            crash program (Error 001:199)
          ends
```

TRACE OF EXAMPLE TWO USING THE EXAMPLE TRAP HANDLER

It is extremely educational, and somewhat interesting to watch the OS-9 user debugger trace through the execution of example two (using the example trap handler). User trap handlers appear to the debugger exactly as subroutines, so tracing through them is possible. If this is done, the output should look something like this:

```
(beginning of second example program)
PC: Test          >4E45             trap #5
```

NOTE: Because the trap handler has not been linked as in EXAMPLE ONE, control jumps to the subroutine LinkTrap:

```
PC: LinkTrap      >508F             addq.l #8,a7
PC: LinkTrap+2    >48E7C0E0         movem.l d0-d1/a0-a2,-(a7)
PC: LinkTrap+6    >7005             moveq.l #5,d0
PC: LinkTrap+8    >7200             moveq.l #0,d1
PC: LinkTrap+A    >41FAFFDC         lea.l bname+4(pc),a0
PC: LinkTrap+E    >4E400021         os9 F$TLink
```

NOTE: Control switches to the subroutine TrapInit and then returns to LinkTrap:

```
PC: etext+1168D0  >4CDFC000         movem.l (a7)+,a6-a7
PC: etext+1168D4  >4E75             rts
PC: LinkTrap+12   >65E8             bcs.s Test+E
PC: LinkTrap+14   >4CDF0703         movem.l (a7)+,d0-d1/a0-a2
PC: LinkTrap+18   >5997             subq.l #4,(a7)
PC: LinkTrap+1A   >4E75             rts
```

TRACE OF EXAMPLE TWO (continued):

NOTE: Control now returns to the main program to re-execute the tcall instruction.

```
PC: Test          >4E45          trap #5
PC: etext+1168D6  >48E7C080      movem.l d0-d1/a0,-(a7)
PC: etext+1168DA  >302F0010      move.w 16(a7),d0
PC: etext+1168DE  >B07C0001      cmp.w #1,d0
PC: etext+1168E2  >621C          bhi.s etext+11E900
PC: etext+1168E4  >6706          beq.s etext+11E8EC
PC: etext+1168E6  >41FA0026      lea.l etext+11E90E(pc),a0
PC: etext+1168EA  >6004          bra.s etext+11E8F0
PC: etext+1168F0  >7001          moveq.l #1,d0
PC: etext+1168F2  >7250          moveq.l #80,d1
PC: etext+1168F4  >4E40008C      os9 I$WritLn
Microware Systems Corporation
PC: etext+1168F8  >650A          bcs.s etext+11E904
PC: etext+1168FA  >4CDFC103      movem.l (a7)+,d0-d1/a0/a6-a7
PC: etext+1168FE  >4E75          rts
PC: Test+4        >6508          bcs.s Test+E
PC: Test+6        >4E45          trap #5
PC: etext+1168D6  >48E7C080      movem.l d0-d1/a0,-(a7)
PC: etext+1168DA  >302F0010      move.w 16(a7),d0
PC: etext+1168DE  >B07C0001      cmp.w #1,d0
PC: etext+1168E2  >621C          bhi.s etext+11E900
PC: etext+1168E4  >6706          beq.s etext+11E8EC->
PC: etext+1168EC  >41FA003F      lea.l etext+11E92D(pc),a0
PC: etext+1168F0  >7001          moveq.l #1,d0
PC: etext+1168F2  >7250          moveq.l #80,d1
PC: etext+1168F4  >4E40008C      os9 I$WritLn
Quality keeps us #1
PC: etext+1168F8  >650A          bcs.s etext+11E904
PC: etext+1168FA  >4CDFC103      movem.l (a7)+,d0-d1/a0/a6-a7
PC: etext+1168FE  >4E75          rts
PC: Test+A        >6502          bcs.s Test+E
PC: Test+C        >7200          moveq.l #0,d1
PC: Test+E        >4E400006      os9 F$Exit
```

End of Chapter 13

THE MATH MODULE

STANDARD FUNCTION LIBRARY MODULE

OS-9/68000 is provided with a module which contains common subroutines for extended mathematical and I/O conversion functions. This module is also used by the OS-9 C, Basic09 and Fortran compilers.

Normally, the standard function module provides extended functions using software routines. The software based modules can be easily replaced by modules that utilize arithmetic processing hardware without requiring any alteration of application software. The standard function module is designed to be installed as user trap routines and is called using the 68000 TRAP instruction. A library file containing the module is a standard part of OS-9/68000. This library may be called directly by user programs in non-OS-9 target systems without using the TRAP call.

The module names, functions and corresponding user trap numbers are shown below:

Trap Module	Library Module	Trap#	Function
Math	Math.l	15	Basic floating point math, extended integer math and type conversion. Transcendental and extended mathematical functions.

CALLING STANDARD FUNCTION MODULE ROUTINES

The standard function library module can be pre-loaded in memory for quick access when needed (using the OS-9 LOAD command). This can be made part of the system's STARTUP file. It is not recommended to include the trap handlers in the OS9BOOT file. The following description of standard function module linkage and calling methods is intended for assembly language programmers. Programs generated by the OS-9 compilers automatically perform all required functions without any special action on the part of the user.

Prior to calling the standard function modules, an assembly language program should use the OS-9 "F$TLink" system call using as parameters the trap numbers and module names given in the above table. This will install and link the user's process to the desired module(s). Calls to individual routines are made using the TRAP instruction. For example, a call to the FADD function would look like this:

```
trap    #T$Math         Trap number of module
dc.w    T$FAdd          Code of FAdd function
```

For simplicity and ease of reading, a macro has been included in the assembler for this purpose. The following line is equivalent to the above example:

```
tcall   T$Math,T$FAdd  Trap number and code for FAdd
```

In non-OS-9 target environments, these routines may also be called directly using BSR instructions. For example:

```
bsr     _T$FAdd        Floating point addition.
```

Many functions set the MPU status register N, Z, V and C bits so the trap or bsr may be immediately followed by a conditional branch instruction for comparisons and error checking. When an error occurs, the system-wide convention is followed where the C condition code bit will be set and D1 will return the specific error code.

In some cases a trapv instruction will be executed at the end of a function. This will cause a trapv exception if the V (overflow) condition code is set.

DATA FORMATS

Two integer types are supported by some functions:

```
unsigned        32-bit unsigned integers
long            32-bit signed integers
```

Two floating point formats are supported:

```
float           32-bit floating point numbers
double          64-bit double precision floating point numbers
```

The floating point math routines use formats based on the proposed IEEE standard for compatibility with floating point math hardware. 32-bit floating point operands are internally converted to 64-bit double precision before computation and converted back to 32 bits afterwards as required by the IEEE and C language standards. Therefore, the float type has no speed advantage over the double type. De-normalized numbers and negative zero are not supported in this package.

THE MATH MODULE

This module provides single and double precision floating point arithmetic, extended integer arithmetic and type conversion routines.

Integer Operations

T$LMul T$UMul T$LDiv T$LMod T$UDiv T$UMod

Single Precision Floating Point Operations

T$FAdd T$FInc T$FSub T$FDec T$FMul T$FDiv T$FCmp T$FNeg

Double Precision Floating Point Operations

T$DAdd T$DInc T$DSub T$DDec T$DMul T$DDiv T$DCmp T$DNeg

ASCII to Numeric Conversions

T$AtoN T$AtoL T$AtoU T$AtoF T$AtoD

Numeric to ASCII Conversions

T$LtoA T$UtoA T$FtoA T$DtoA

Numeric to Numeric Conversions

T$LtoF T$LtoD T$UtoF T$UtoD T$FtoL T$DtoL T$FtoU T$DtoU
T$FtoD T$DtoF T$FTrn T$DTrn T$FInt T$DInt T$DNrm

The math module also provides transcendental and extended mathematical functions. The precision of these routines is controlled by the calling routine. For example if fourteen digits of precision are required, the floating-point representation for 1E-014 should be passed to the routine.

Function Name	Operation
T$Sin	Sine function
T$Cos	Cosine function
T$Tan	Tangent function
T$Asn	Arc sine function
T$Acs	Arc cosine function
T$Atn	Arc tangent function
T$Log	Natural logarithm function
T$Log10	Common logarithm function
T$Sqrt	Square root function
T$Exp	Exponential function
T$Power	Power function

The following table contains the hex representations which should be passed to these routines to define the precision of the operation.

Precision	Hex Representation
1E-001	3fb99999 9999999a
1E-002	3f847ae1 47ae147b
1E-003	3f50624d d2f1a9fc
1E-004	3f1a36e2 eb1c432d
1E-005	3ee4f8b5 88e368f1
1E-006	3eb0c6f7 a0b5ed8e
1E-007	3e7ad7f2 9abcaf4a
1E-008	3e45798e e2308c3b
1E-009	3e112e0b e826d696
1E-010	3ddb7cdf d9d7bdbd
1E-011	3da5fd7f e1796497
1E-012	3d719799 812dea12
1E-013	3d3c25c2 68497683
1E-014	3d06849b 86a12b9c

NOTE: Using a precision greater than 14 digits may cause the routine to get trapped in an infinite loop.

T$Acs **ArcCosine Function** **T$Acs**

ASSEMBLER CALL: TCALL T$Math,T$Acs

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = ArcCos(x) (in radians)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the arc cosine() in radians. If the operand passed is illegal an error will be returned.

T$Asn **ArcSine Function** **T$Asn**

ASSEMBLER CALL: TCALL T$Math,T$Asn

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = ArcSin(x) (in radians)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the arcsine() in radians. If the operand passed is illegal an error will be returned.

T$Atn **ArcTangent Function** **T$Atn**

ASSEMBLER CALL: TCALL T$Math,T$Atn

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = ArcTan(x) (in radians)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the arc tangent() in radians. If the operand passed is illegal an error will be returned.

T$AtoD **ASCII to Double-Precision Floating-Point** **T$AtoD**

ASSEMBLER CALL: TCALL T$Math,T$AtoD

INPUT: (a0) = Pointer to ASCII string
Format: <sign><digits>.<digits><E or e><sign><digits>

OUTPUT: (a0) = Updated pointer
d0:d1 = Double-precision floating-point number

POSSIBLE ERRORS: E$NotNum, or E$FmtErr

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on underflow or overflow.
C Set on error.

FUNCTION: This function performs a conversion from an ASCII string to a double-precision floating-point number. If the first character is not the sign (+ or -) or a digit, E$NotNum is returned. If the first character following the "E" is not the sign or a digit, E$FmtErr is returned.

If the overflow bit (V) is set, zero (on underflow) or +/- infinity (overflow) is returned.

T$AtoF **ASCII to Single-Precision Floating-Point** **T$AtoF**

ASSEMBLER CALL: TCALL T$Math,T$AtoF

INPUT: (a0) = Pointer to ASCII string
Format: <sign><digits>.<digits><E or e><sign><digits>

OUTPUT: (a0) = Updated pointer
d0.l = Single-precision floating-point number

POSSIBLE ERRORS: E$NotNum, or E$FmtErr

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on underflow or overflow
C Set on error.

FUNCTION: This function performs a conversion from an ASCII string to a single-precision floating-point number. If the first character is not the sign (+ or -) or a digit, E$NotNum is returned. If the first character following the "E" is not the sign or a digit, E$FmtErr is returned. If the overflow bit (V) is set, zero (on underflow) or +/- infinity (overflow) is returned.

T$AtoL **ASCII to Long Conversion** **T$AtoL**

ASSEMBLER CALL: TCALL T$Math,T$AtoL

INPUT: (a0) = Pointer to ASCII string (format: <sign><digits>)

OUTPUT: (a0) = Updated pointer
d0.l = Signed long

POSSIBLE ERRORS: E$NotNum

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on overflow.
C Set on error.

FUNCTION: This function performs a conversion from an ASCII string to a signed long integer. If the first character is not a sign (+ or -) or a digit, an error is returned.

T$AtoN **ASCII to Numeric Conversion** **T$AtoN**

ASSEMBLER CALL: TCALL T$Math,T$AtoN

INPUT: (a0) = Pointer to ASCII string

OUTPUT: (a0) = Updated pointer
d0 = Number if returned as long (signed or unsigned)
d0:d1 = Number if returned in floating point format

POSSIBLE ERRORS: TrapV

CONDITION CODES: See explanation below

FUNCTION: This function can return results of various types depending on the format of the input string and the magnitude of the converted value. The type of the result is passed back to the calling program using the V and N condition code bits.

V=0 and N=1 indicates a signed integer is returned in d0.l
V=0 and N=0 indicates an unsigned integer is returned in d0.l
V=1 indicates a double-precision number is returned in d0:d1

If any of the following conditions are met the number will be returned as a double-precision floating-point value:

- The number is positive and overflows an unsigned long.
- The number is negative and overflows a signed long.
- The number contains a decimal point and/or an "E" exponent.

If none of the above conditions are met, the result will be returned as an unsigned long (if positive) or a signed long (if negative).

T$AtoU **ASCII to Unsigned Conversion** **T$AtoU**

ASSEMBLER CALL: TCALL T$Math,T$AtoU

INPUT: (a0) = Pointer to ASCII string (format: <digits>)

OUTPUT: (a0) = Updated pointer
d0.l = Unsigned long

POSSIBLE ERRORS: E$NotNum

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on overflow.
C Set on error.

FUNCTION: This function performs a conversion from an ASCII string to an unsigned long integer. If the first character is not a digit, an error is returned.

T$Cos **Cosine Function** **T$Cos**

ASSEMBLER CALL: TCALL T$Math,T$Cos

INPUT: d0:d1 = x (in radians)
d2:d3 = Precision

OUTPUT: d0:d1 = Cos(x)

POSSIBLE ERRORS: None

CONDITION CODES: C Always clear.

FUNCTION: This function returns the cosine() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$DAdd **Double Precision Addition** **T$DAdd**

ASSEMBLER CALL: TCALL T$Math,T$DAdd

INPUT: d0:d1 = Addend
d2:d3 = Augend

OUTPUT: d0:d1 = Result (d0:d1 + d2:d3)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow or overflow.
C Always cleared.

FUNCTION: This function adds two double-precision floating point numbers. Overflow and Underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$DCmp **Double Precision Compare** **T$DCmp**

ASSEMBLER CALL: TCALL T$Math,T$DCmp

INPUT: d0:d1 = First operand
d2:d3 = Second operand

OUTPUT: d0.l through d3.l remain unchanged

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if second operand is larger than the first.
Z Set if operands are equal.
V Always cleared.
C Always cleared.

FUNCTION: Two double-precision floating point numbers are compared by this function. The operands passed to this function are not destroyed.

T$DDec **Double Precision Decrement** **T$DDec**

ASSEMBLER CALL: TCALL T$Math,T$DDec

INPUT: d0:d1 = Operand

OUTPUT: d0:d1 = Result (d0:d1 - 1.0)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Set if result is negative.
- Z Set if result is zero.
- V Set on underflow.
- C Always cleared.

FUNCTION: This function subtracts 1.0 from the double-precision floating point operand. Underflow is indicated by the V bit being set. If an underflow occurs, a trapv exception will be generated and zero will be returned.

T$DDiv **Double Precision Divide** **T$DDiv**

ASSEMBLER CALL: TCALL T$Math,T$DDiv

INPUT: d0:d1 = Dividend
d2:d3 = Divisor

OUTPUT: d0:d1 = Result (d0:d1 / d2:d3)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Set if result is negative.
- Z Set if result is zero.
- V Set on underflow, overflow or divide by zero.
- C Set on divide by zero

FUNCTION: This function performs division on two double-precision floating point numbers. Overflow, underflow, and divide-by-zero are indicated by the V bit being set. In any case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow or divide-by-zero, infinity (with the proper sign) will be returned.

T$DInc **Double Precision Increment** **T$DInc**

ASSEMBLER CALL: TCALL T$Math,T$DInc

INPUT: d0:d1 = Operand

OUTPUT: d0:d1 = Result (d0:d1 + 1.0)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Set if result is negative.
- Z Set if result is zero.
- V Set on overflow.
- C Always cleared.

FUNCTION: This function adds 1.0 to the double-precision floating point operand. Overflow is indicated by the V bit being set. If an overflow occurs, a trapv exception will be generated and infinity (with the proper sign) will be returned.

T$DInt **Round Double-Precision Floating-Point Number** **T$DInt**

ASSEMBLER CALL: TCALL T$Math,T$DInt

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0:d1 = Rounded double-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: Floating point numbers consist of two parts: integer part and fractional part. The purpose of this function is to round the floating point number passed to it, leaving only an integer part. If the fractional part is exactly 0.5 then the integer part is rounded to an even number.

EXAMPLES:
- 23.45 rounds to 23.00
- 23.50 rounds to 24.00
- 23.73 rounds to 24.00
- 24.50 rounds to 24.00 (rounds to even number)

T$DMul **Double Precision Multiplication** **T$DMul**

ASSEMBLER CALL: TCALL T$Math,T$DMul

INPUT: d0:d1 = Multiplicand
d2:d3 = Multiplier

OUTPUT: d0:d1 = Result (d0:d1 * d2:d3)

POSSIBLE ERRORS: TrapV

CONDITION CODES:

N	Set if result is negative.
Z	Set if result is zero.
V	Set on underflow or overflow.
C	Always cleared.

FUNCTION: This function multiplies two double-precision floating point numbers. Overflow and underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$DNeg **Double Precision Negate** **T$DNeg**

ASSEMBLER CALL: TCALL T$Math,T$DNeg

INPUT: d0:d1 = Operand

OUTPUT: d0:d1 = Result (d0:d1 * -1.0)

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Always cleared.
C Always cleared.

FUNCTION: This function negates a double-precision floating point operand. In order to eliminate the overhead of calling this routine, it is very simple to just change the sign bit of the floating-point number. The only case you have to watch out for is when the number is zero. This is because negative zero is not supported by this package.

This example is written as a subroutine and expects the floating-point number to be in d0:d1.

```
Negate  tst.l d0      test for zero
        beq.s Neg10   branch if it is zero
        bchg  #31,d0  change sign bit
Neg10   rts           return
```

T$DNrm **64-bit Unsigned to Double-Precision Conversion** **T$DNrm**

ASSEMBLER CALL: TCALL T$Math,T$DNrm

INPUT: d0:d1 = 64-bit Unsigned Integer
d2.1 = Exponent

OUTPUT: d0:d1 = Double-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on underflow or overflow.
C Undefined.

FUNCTION: Double precision floating point numbers maintain 52 bits of mantissa. This function converts a 64 bit binary number to double precision format. The extra 12 bits are rounded. If an underflow or overflow occurs, the V bit is set but a trapv exception is not generated.

T$DSub **Double Precision Subtraction** **T$DSub**

ASSEMBLER CALL: TCALL T$Math,T$DSub

INPUT: d0:d1 = Minuend
d2:d3 = Subtrahend

OUTPUT: d0:d1 = Result (d0:d1 - d2:d3)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow or overflow.
C Always cleared.

FUNCTION: This function performs subtraction on two double-precision floating point numbers. Overflow and underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$DtoA **Double-Precision Floating-Point to ASCII** **T$DtoA**

ASSEMBLER CALL: TCALL T$Math,T$DtoA

INPUT: d0:d1 = Double-precision floating-point number
d2.l = Low-Word: digits desired in result
High-Word: digits desired after decimal-point
(a0) = Pointer to conversion buffer

OUTPUT: (a0) = ASCII digit string
d0.l = Two's complement exponent

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if the number is negative.
Z Undefined.
V Undefined.
C Undefined.

FUNCTION: The double-precision float passed to this function is converted to an ASCII string. The conversion terminates as soon as the number of digits requested have been converted or when the specified digit after the decimal point has been reached; whichever comes first. A null will be appended to the end of the string. Therefore the buffer should be one (1) byte larger than the expected number of digits.

The converted string will only contain the mantissa digits. The sign of the number is indicated by the N bit, and the exponent is returned in register d0.

T$DtoF **Double to Single Floating-Point Conversion** **T$DtoF**

ASSEMBLER CALL: TCALL T$Math,T$DtoF

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0.1 = Single-precision floating-point number

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Undefined.
- Z Undefined.
- V Set on underflow or overflow.
- C Undefined.

FUNCTION: This function converts floating-point numbers in double-precision format to single-precision format. No errors are possible and all condition codes are undefined. If an overflow or underflow occurs, the V bit will be set and a trapv exception will be generated.

T$DtoL **Double-Precision to Signed Long Integer** **T$DtoL**

ASSEMBLER CALL: TCALL T$Math,T$DtoL

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0.1 = Signed Long Integer

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Undefined.
- Z Undefined.
- V Set on overflow.
- C Undefined.

FUNCTION: The integer portion of the floating point number is converted to a signed long integer. The fraction is truncated. If an overflow occurs, the V bit will be set and a trapv exception will be generated.

T$DtoU **Double-Precision to Unsigned Long Integer** **T$DtoU**

ASSEMBLER CALL: TCALL T$Math,T$DtoU

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0.l = Unsigned Long Integer

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Undefined.
Z Undefined.
V Set on overflow.
C Undefined.

FUNCTION: The integer portion of the floating point number is converted to an unsigned long integer. The fraction is truncated. If an overflow occurs, the V bit will be set and a trapv exception will be generated.

T$DTrn **Truncate Double-Precision Floating-Point Number** **T$DTrn**

ASSEMBLER CALL: TCALL T$Math,T$DTrn

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0:d1 = Normalized integer portion of the floating point number
d2:d3 = Normalized fractional portion of the floating point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: Floating point numbers consist of two parts: integer part and fractional part. The purpose of this function is to separate the two parts. For example if the number passed is 283.75 this function will return 283.00 in d0:d1 and 0.75 in d2:d3

T$Exp **Exponential Function** **T$Exp**

ASSEMBLER CALL: TCALL T$Math,T$Exp

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = Exp(x)

POSSIBLE ERRORS: None

CONDITION CODES: C Always clear.

FUNCTION: This function performs the exponential function on the argument passed. That is, it raises e to the x power (where e = 2.718282 and x is the argument passed).

T$FAdd **Single Precision Addition** **T$FAdd**

ASSEMBLER CALL: TCALL T$Math,T$FAdd

INPUT: d0.l = Addend
d1.l = Augend

OUTPUT: d0.l = Result (d0 + d1)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow or overflow.
C Always cleared.

FUNCTION: This function adds two single-precision floating point numbers. Overflow and underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$FCmp **Single Precision Compare** **T$FCmp**

ASSEMBLER CALL: TCALL T$Math,T$FCmp

INPUT: d0.l = First operand
d1.l = Second operand

OUTPUT: d0.l and d1.l remain unchanged

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if second operand is larger than the first.
Z Set if operands are equal.
V Always cleared.
C Always cleared.

FUNCTION: Two single-precision floating point numbers are compared by this function. The operands passed to this function are not destroyed.

T$FDec **Single Precision Decrement** **T$FDec**

ASSEMBLER CALL: TCALL T$Math,T$FDec

INPUT: d0.l = Operand

OUTPUT: d0.l = Result (d0 - 1.0)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow.
C Always cleared.

FUNCTION: This function subtracts 1.0 from the single-precision floating point operand. Underflow is indicated by the V bit being set. If an underflow occurs, a trapv exception will be generated and zero will be returned.

T$FDiv **Single Precision Divide** **T$FDiv**

ASSEMBLER CALL: TCALL T$Math,T$FDiv

INPUT: d0.l = Dividend
d1.l = Divisor

OUTPUT: d0.l = Result (d0 / d1)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow, overflow or divide by zero.
C Set on divide by zero.

FUNCTION: This function performs division on two single-precision floating point numbers. Overflow, underflow and divide-by-zero are indicated by the V bit being set. In any case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow or divide-by-zero, infinity (with the proper sign) will be returned.

T$FInc **Single Precision Increment** **T$FInc**

ASSEMBLER CALL: TCALL T$Math,T$FInc

INPUT: d0.l = Operand

OUTPUT: d0.l = Result (d0 + 1.0)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on overflow.
C Always cleared.

FUNCTION: This function adds 1.0 to the single-precision floating point operand. Overflow is indicated by the V bit being set. If an overflow occurs, a trapv exception will be generated and infinity (with the proper sign) will be returned.

T$FInt **Round Single-Precision Floating-Point Number** **T$FInt**

ASSEMBLER CALL: TCALL T$Math,T$FInt

INPUT: d0.l = Single-precision floating-point number

OUTPUT: d0.l = Rounded single-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: Floating point numbers consist of two parts: integer part and fractional part. The purpose of this function is to round the floating point number passed to it, leaving only an integer part. If the fractional part is exactly 0.5 then the integer part is rounded to an even number.

EXAMPLES:
```
23.45 rounds to 23.00
23.50 rounds to 24.00
23.73 rounds to 24.00
24.50 rounds to 24.00 (rounds to even number)
```

T$FMul **Single Precision Multiplication** **T$FMul**

ASSEMBLER CALL: TCALL T$Math,T$FMul

INPUT: d0.l = Multiplicand
d1.l = Multiplier

OUTPUT: d0.l = Result (d0 * d1)

POSSIBLE ERRORS: TrapV

CONDITION CODES:
```
N  Set if result is negative.
Z  Set if result is zero.
V  Set on underflow or overflow.
C  Always cleared.
```

FUNCTION: This function multiplies two single-precision floating point numbers. Overflow and underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$FNeg **Single Precision Negate** **T$FNeg**

ASSEMBLER CALL: TCALL T$Math,T$FNeg

INPUT: d0.l = Operand

OUTPUT: d0.l = Result (d0 * -1.0)

POSSIBLE ERRORS: None

CONDITION CODES:

N	Set if result is negative.
Z	Set if result is zero.
V	Always cleared.
C	Always cleared.

FUNCTION: This function negates a single-precision floating point operand. In order to eliminate the overhead of calling this routine, it is very simple to just change the sign bit of the floating-point number. The only case you have to watch out for is when the number is zero. This is because negative zero is not supported by this package.

This example is written as a subroutine and expects the floating-point number to be in d0.

```
Negate  tst.l d0      test for zero
        beq.s Negl0   branch if it is zero
        bchg  #31,d0  change sign bit
Negl0   rts           return
```

T$FSub **Single Precision Subtraction** **T$FSub**

ASSEMBLER CALL: TCALL T$Math,T$FSub

INPUT: d0.l = Minuend
d1.l = Subtrahend

OUTPUT: d0.l = Result (d0 - d1)

ERROR OUTPUT: TrapV

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on underflow or overflow.
C Always cleared.

FUNCTION: This function performs subtraction on two single-precision floating point numbers. Overflow and underflow are indicated by the V bit being set. In either case, a trapv exception will be generated. If an underflow caused the exception, zero will be returned. If it was an overflow, infinity (with the proper sign) will be returned.

T$FtoA **Single-Precision Floating-Point to ASCII** **T$FtoA**

ASSEMBLER CALL: TCALL T$Math,T$FtoA

INPUT:
```
d0.l = Single-precision floating-point number
d2.l = Low-Word: digits desired in result
       High-Word: digits desired after decimal-point
(a0) = Pointer to conversion buffer
```

OUTPUT:
```
(a0) = ASCII digit string
d0.l = Two's complement exponent
```

POSSIBLE ERRORS: None

CONDITION CODES:
```
N  Set if the number is negative.
Z  Undefined.
V  Undefined.
C  Undefined.
```

FUNCTION: The single-precision float passed to this function is converted to an ASCII string. The conversion terminates as soon as the number of digits requested have been converted or when the specified digit after the decimal point has been reached; whichever comes first. A null will be appended to the end of the string. Therefore the buffer should be one (1) byte larger than the expected number of digits.

The converted string will only contain the mantissa digits. The sign of the number is indicated by the N bit, and the exponent is returned in register d0.

T$FtoD **Single to Double Floating-Point Conversion** **T$FtoD**

ASSEMBLER CALL: TCALL T$Math,T$FtoD

INPUT: d0.l = Single-precision floating-point number

OUTPUT: d0:d1 = Double-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: This function converts floating-point numbers in single-precision format to double-precision format. No errors are possible and all condition codes are undefined.

T$FtoL **Single-Precision to Signed Long Integer** **T$FtoL**

ASSEMBLER CALL: TCALL T$Math,T$FtoL

INPUT: d0.l = Single-precision floating-point number

OUTPUT: d0.l = Signed Long Integer

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Undefined.
- Z Undefined.
- V Set on overflow.
- C Undefined.

FUNCTION: The integer portion of the floating point number is converted to a signed long integer. The fraction is truncated. If an overflow occurs, the V bit will be set and a trapv exception will be generated.

T$FtoU **Single-Precision to Unsigned Long Integer** **T$FtoU**

ASSEMBLER CALL: TCALL T$Math,T$FtoU

INPUT: d0.l = Single-precision floating-point number

OUTPUT: d0.l = Unsigned Long Integer

POSSIBLE ERRORS: TrapV

CONDITION CODES:
- N Undefined.
- Z Undefined.
- V Set on overflow.
- C Undefined.

FUNCTION: The integer portion of the floating point number is converted to an unsigned long integer. The fraction is truncated. If an overflow occurs, the V bit will be set and a trapv exception will be generated.

T$FTrn **Truncate Single-Precision Floating-Point Number** **T$FTrn**

ASSEMBLER CALL: TCALL T$Math,T$FTrn

INPUT: d0.l = Single-precision floating-point number

OUTPUT: d0.l = Truncated single-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: Floating point numbers consist of two parts: integer part and fractional part. The purpose of this function is to truncate the fractional part. For example if the number passed is 283.75 this function will return 283.00.

T$LDiv **Long (Signed) Divide** **T$LDiv**

ASSEMBLER CALL: TCALL T$Math,T$LDiv

INPUT: d0.l = Dividend
d1.l = Divisor

OUTPUT: d0.l = Result (d0 / d1)

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if result is negative.
Z Set if result is zero.
V Set on divide by zero.
C Always cleared.

FUNCTION: This function performs 32 bit integer division. A division by zero error is indicated by the overflow bit being set. If a division by zero is attempted, infinity (with the proper sign) will be returned.

Positive Infinity = $7FFFFFFF
Negitive Infinity = $80000000

T$LMod **Long (Signed) Modulus** **T$LMod**

ASSEMBLER CALL: TCALL T$Math,T$LMod

INPUT: d0.l = Dividend
d1.l = Divisor

OUTPUT: d0.l = Result (Mod(d0/d1))

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if result is negitive.
Z Set if result is zero.
V Set on divide by zero.
C Always cleared.

FUNCTION: The remainder (modulo) of the integer division is returned by this function. If an overflow occurs, the V bit will be set and zero will be returned.

T$LMul **Long (Signed) Multiply** **T$LMul**

ASSEMBLER CALL: TCALL T$Math,T$LMul

INPUT: d0.l = Multiplicand
d1.l = Multiplier

OUTPUT: d0.l = Result (d0 * d1)

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if result is negitive.
Z Set if result is zero.
V Set on overflow.
C Always cleared.

FUNCTION: This function performs a 32 bit signed integer multiplication. If an overflow occurs, the V bit will be set and the lower 32 bits of the result will be returned. If an overflow occurs, the sign of the result will still be correct.

T$Log **Natural Logarithm Function** **T$Log**

ASSEMBLER CALL: TCALL T$Math,T$Log

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = Log(x)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the natural logarithm of the argument passed. If an illegal argument is passed an error will be returned.

T$Log10 **Common Logarithm Function** **T$Log10**

ASSEMBLER CALL: TCALL T$Math,T$Log10

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = Log10(x)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the common logarithm of the argument passed. If an illegal argument is passed an error will be returned.

T$LtoA **Signed Integer to ASCII Conversion** **T$LtoA**

ASSEMBLER CALL: TCALL T$Math,T$LtoA

INPUT: d0.l = Signed long integer
(a0) = Pointer to conversion buffer

OUTPUT: (a0) = ASCII digit string

POSSIBLE ERRORS: None

CONDITION CODES:
N Set if the number is negative.
Z Undefined.
V Undefined.
C Undefined.

FUNCTION: The signed long passed to this function is converted to an ASCII string of ten (10) digits. If the number is smaller than ten digits, it is right justified and padded with leading zeros. A null is appended to the end of the string making the minimum size of the buffer eleven (11) characters.

NOTE: The sign is indicated by the N bit and is not included in the ASCII string.

T$LtoD **Signed Long to Double-Precision Floating-Point** **T$LtoD**

ASSEMBLER CALL: TCALL T$Math,T$LtoD

INPUT: d0.l = Signed long integer

OUTPUT: d0:d1 = Double-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: The signed integer is converted to a double-precision float by this function. No errors are possible and all condition codes are undefined.

T$LtoF **Signed Long to Single-Precision Floating-Point** **T$LtoF**

ASSEMBLER CALL: TCALL T$Math,T$LtoF

INPUT: d0.l = Signed long integer

OUTPUT: d0.l = Single-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: The signed integer is converted to a single-precision float by this function. No errors are possible and all condition codes are undefined.

T$Power **Power Function** **T$Power**

ASSEMBLER CALL: TCALL T$Math,T$Power

INPUT: d0:d1 = x
d2:d3 = y
d4:d5 = Precision

OUTPUT: d0:d1 = x^y

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function performs the power function on the arguments passed. That is, it raises x to the y power. If an illegal argument is passed an error will be returned.

T$Sin **Sine Function** **T$Sin**

ASSEMBLER CALL: TCALL T$Math,T$Sin

INPUT: d0:d1 = x (in radians)
d2:d3 = Precision

OUTPUT: d0:d1 = Sin(x)

POSSIBLE ERRORS: None

CONDITION CODES: C Always clear.

FUNCTION: This function returns the sine() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$Sqr **Square Root Function** **T$Sqrt**

ASSEMBLER CALL: TCALL T$Math,T$Sqrt

INPUT: d0:d1 = x
d2:d3 = Precision

OUTPUT: d0:d1 = Sqrt(x)

POSSIBLE ERRORS: E$IllArg

CONDITION CODES: C Set on error.

FUNCTION: This function returns the square root of the argument passed. If an illegal argument is passed an error will be returned.

T$Tan **Tangent Function** **T$Tan**

ASSEMBLER CALL: TCALL T$Math,T$Tan

INPUT: d0:d1 = x (in radians)
d2:d3 = Precision

OUTPUT: d0:d1 = Tan(x)

POSSIBLE ERRORS: None

CONDITION CODES: C Always clear.

FUNCTION: This function returns the tangent() of an angle. The angle must be specified in radians. No errors are possible, and all condition codes are undefined.

T$UDiv **Unsigned Divide** **T$UDiv**

ASSEMBLER CALL: TCALL T$Math,T$UDiv

INPUT: d0.l = Dividend
d1.l = Divisor

OUTPUT: d0.l = Result (d0 / d1)

POSSIBLE ERRORS: None

CONDITION CODES:
N Undefined.
Z Set if result is zero.
V Set on divide by zero.
C Always cleared.

FUNCTION: This function performs 32 bit unsigned integer division. Division by zero error is indicated by the overflow bit being set. If a division by zero is attempted, infinity ($FFFFFFFF) will be returned.

T$UMod **Unsigned Modulus** **T$UMod**

ASSEMBLER CALL: TCALL T$Math,T$UMod

INPUT: d0.l = Dividend
d1.l = Divisor

OUTPUT: d0.l = Result (Mod(d0/d1))

POSSIBLE ERRORS: None

CONDITION CODES:
N Undefined.
Z Set if result is zero.
V Set on divide by zero.
C Always cleared.

FUNCTION: The remainder (modulo) of the integer division is returned by this function. If an overflow occurs, the V bit will be set and zero will be returned.

T$UMul **Unsigned Multiply** **T$UMul**

ASSEMBLER CALL: TCALL T$Math,T$UMul

INPUT: d0.l = Multiplicand
d1.l = Multiplier

OUTPUT: d0.l = Result (d0 * d1)

POSSIBLE ERRORS: None

CONDITION CODES:
N Undefined.
Z Set if result is zero.
V Set on overflow.
C Always cleared.

FUNCTION: This function performs a 32 bit unsigned integer multiplication. If an overflow occurs, the V bit will be set and the lower 32 bits of the result will be returned.

T$UtoA **Unsigned Integer to ASCII Conversion** **T$UtoA**

ASSEMBLER CALL: TCALL T$Math,T$UtoA

INPUT: d0.l = Unsigned long integer
(a0) = Pointer to conversion buffer

OUTPUT: (a0) = ASCII digit string

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: The unsigned long passed to this function is converted to an ASCII string of ten (10) digits. If the number is smaller than ten digits, it is right justified and padded with leading zeros. A null is appended to the end of the string making the minimum size of the buffer eleven (11) characters.

T$UtoD **Unsigned Long to Double-Precision Floating-Point** **T$UtoD**

ASSEMBLER CALL: TCALL T$Math,T$UtoD

INPUT: d0.l = Unsigned long integer

OUTPUT: d0:d1 = Double-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: The unsigned integer is converted to a double-precision float by this function. No errors are possible and all condition codes are undefined.

T$UtoF **Unsigned Long to Single-Precision Floating-Point** **T$UtoF**

ASSEMBLER CALL: TCALL T$Math,T$UtoF

INPUT: d0.l = Unsigned long integer

OUTPUT: d0.l = Single-precision floating-point number

POSSIBLE ERRORS: None

CONDITION CODES: All condition codes are undefined.

FUNCTION: The unsigned integer is converted to a single-precision float by this function. No errors are possible and all condition codes are undefined.

End of Chapter 14

EVENTS

In certain application programs, a shared system resource must be protected from being accessed simultaneously by several concurrent processes. For example, if two processes need to communicate with each other through a shared data module, it may be necessary to synchronize the processes so that only one updates the data module at any given time. This can be done with an event.

Events are another form of inter-process communication provided by OS-9, along with signals, pipelines and data modules. Pipes and data modules deal with the sharing of data between any number of processes. Events and signals, are concerned with the passing of control information among processes. Neither is intended for the transmission of data, although signals do send a single word of data from the signalling to the receiving process, which can be used to represent and information capable of being represented in this amount of space. Events do not transmit any information, although processes using the event system may obtain information about the event, and conceivably could use it in some fashion other than as a signaling mechanism.

An event is a special type of system global variable. It is used as a tool to synchronize processes. Events are 32 byte data structures maintained by the system. Elements of the structure include a unique ID number, a twelve character (maximum) name, an integer event value, increments which are applied when the event is awaited or signaled, a link count and pointers to the prior and next events in the queue, if any.

The OS-9 event system provides several capabilities. These include facilities to create and delete events, to permit processes to link and unlink events, obtain information about them, suspend operation until an event occurs, and various means of signalling. Events are useful for permitting processes to share a common resource without interference or for providing sequencing and execution control of asynchronous processes. As with other OS-9 system services, events may be used directly as service requests in assembly language programs. The Microware C compiler has incorporated them into a series of OS-9 system library functions, for ease of use from within C programs.

Understanding Events

There are two basic operations that may be performed on an event: Wait and Signal. They allow one process to suspend itself while waiting for some event and to reactivate when another process signals the event has occurred.

An event can be thought of as a pigeon hole box for keys at a hotel. The numeric value of the event represents the number of keys in the box. The Signal operation corresponds to checking out of the hotel and returns one key to the box. The Wait operation attempts to remove a key from the box. If none are available, the F$Event manager forces the process to wait. The process waits until a key is available (a Signal operation is performed), removes the key and then continues execution.

For user programs to coordinate sharing a non-re-entrant resource, they must:

1. Wait for the resource to become available.
2. Mark the resource as busy.
3. Use the resource.
4. Signal that the resource is no longer busy.

It is critical for the first two steps in this process to be indivisible, because of timeslicing. Otherwise after the first step, two processes could check the event and find it free. Then each process would try to mark it as busy. This would correspond to two guests being assigned the same room at the hotel. The F$Event service request insures this will not happen by performing both steps in the wait operation.

THE F$EVENT SYSTEM CALL

The F$Event system call provides the mechanism to create named "events" for this type of application. The name "event" was chosen instead of "semaphore" because F$Event provides the flexibility to synchronize processes in a variety of ways not usually found in semaphore primitives. OS-9's event routines are very efficient, and suitable for use in real-time control applications.

Event variables require several maintenence functions as well as the Signal and Wait operations. To keep the number of system calls required to a minimum, all event operations are accessible through the F$Event system call. A function code is passed in register d1.w to indicate which event operation is desired.

Currently, functions exist to allow events to be created, deleted, linked, unlinked and examined. Several variations of the Signal and Wait operations are also provided.

Specific parameters and functions of each event operation are discussed on the following pages. The "Ev$xxx" function names are defined in the system definition file "funcs.a". Actual values for the function codes are resolved by linking with the relocatable library "sys.l" or "usr.l".

Ev$Link **Link to existing event by name** **Ev$Link**

INPUT: (a0) = event name string pointer (max 11-chars)
d1.w = 0 (Ev$Link function code)

OUTPUT: d0.l = event ID number
(a0) = updated past event name

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

POSSIBLE ERRORS: E$BNam - name is syntactically incorrect or > 11 chars
E$EvNF - event not found in the event table

FUNCTION: Ev$Link is used by a program to determine the ID number of an existing event. Once an event has been linked, all subsequent references are made using the event ID returned. This permits the system to access events quickly, while protecting against programs using invalid or deleted events. The event use count is incremented when an Ev$Link is performed. To keep the use count synchronized properly, an Ev$UnLnk should be performed when the event will no longer be used.

Ev$UnLnk **Unlink event** **Ev$UnLnk**

INPUT: d0.l = event ID number
d1.w = 1 (Ev$UnLnk function code)

OUTPUT: None

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION: The unlink function of F$Event is used to inform the system that the event will no longer be used by a process. This causes the event use count to be decremented, and allows the event to be deleted when the count reaches zero. OS-9 uses this only for error checking.

Ev$Creat — **Create new event** — **Ev$Creat**

INPUT:
```
d0.l = initial event variable value
d1.2 = 2 (Ev$Creat function code)
d2.w = auto-increment for Ev$Wait
d3.w = auto-increment for Ev$Signl
(a0) = event name string pointer (max 11-chars)
```

OUTPUT:
```
d0.l = event ID number
(a0) = updated past event name
```

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS:
```
E$BNam - name is syntactically incorrect or > 11 chars
E$EvFull - the event table is full
E$EvBusy - the named event already exists
```

FUNCTION: Events may be created and deleted dynamically as needed. Upon creation, an initial (signed) value is specified, as well as (signed) increments to be applied each time the event occurs or is waited for. The event ID number returned is used in subsequent F$Event calls to refer to the event created.

Ev$Delet — **Delete existing event** — **Ev$Delet**

INPUT:
```
(a0) = event name string pointer (max 11-chars)
d1.w = 3 (Ev$Delet function code)
```

OUTPUT:
```
(a0) = updated past event name
```

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS:
```
E$BNam - name is syntactically incorrect or > 11 chars
E$EvNF - event not found in the event table
E$EvBusy - the event has a non-zero link count
```

FUNCTION: The Ev$Delet function removes an event from the system event table, freeing the entry for use by another event. Events have an implicit use count (initially set to one), which is incremented with each Ev$Link call and decremented with each Ev$UnLnk call. An event may not be deleted unless its use count is zero.

NOTE: OS-9 does not automatically "unlink" events when an F$Exit occurs.

Ev$Wait **Wait for event to occur** **Ev$Wait**

INPUT:
```
d0.l = event ID number
d1.w = 4 (Ev$Wait function code)
d2.l = minimum activation value (signed)
d3.l = maximum activation value (signed)
```

OUTPUT:
```
d1.l = actual event value
```

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION: The Ev$Wait function is used to wait for an event to occur. The event variable is first compared to the range specified in d2 and d3. If the value is not in range, the calling process is suspended in a fi-fo event queue. It waits until an Ev$Signl occurs that puts the value in range. Once the event value is within range, the wait auto-increment (specified at creation) is added to the event variable.

If the process receives a signal while in the event queue, it is activated even though the event has not actually occurred. The auto-increment is not added to the event variable, and the event value returned will not be within the specified range. An error is not returned, but the caller's intercept routine will be executed.

Ev$WaitR **Wait for relative event to occur** **Ev$WaitR**

INPUT:
```
d0.l = event ID number
d1.w = 5 (Ev$WaitR function code)
d2.l = minimum relative activation value (signed)
d3.l = maximum relative activation value (signed)
```

OUTPUT:
```
d1.l = actual event value
d2.l = minimum actual activation value
d3.l = maximum actual activation value
```

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION: The Ev$WaitR function works exactly like Ev$Wait, except that the range specified in d2 and d3 is relative to the current event value. The event value is added to d2 and d3 respectively, and the actual values are returned to the caller. The Ev$Wait function is then executed directly. If an underflow or overflow occurs on the addition, the values $80000000 (minimum integer), and $7fffffff (maximum integer) are used, respectively.

Ev$Read **Read event value without waiting** **Ev$Read**

INPUT:
```
d0.l = event ID number
d1.w = 6 (Ev$Read function code)
```

OUTPUT:
```
d1.l = current event value
```

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION: The Ev$Read function is used to read the value of an event without waiting or effecting the event variable. This can be used to determine the availability of the event (or associated resource) without waiting.

Ev$Info **Return event information** **Ev$Info**

INPUT: d0.l = event index to begin search
d1.w = 7 (Ev$Info function code)
(a0) = ptr to buffer for event information

OUTPUT: d0.l = event index found
(a0) = data returned in buffer

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

POSSIBLE ERRORS: E$EvntID - the index is above all active events

FUNCTION: Ev$Info is an information request function that returns a copy of the 32-byte event table entry associated with an event. Unlike other F$Event functions, only an event index is passed in d0. The index is the system event number, ranging from zero to the maximum number of system events minus one. The event information block for the first active event with an index greater than or equal to this index is returned in the caller's buffer. If none exists, an error is returned. Ev$Info is provided for utilities needing to determine the status of all active events.

Ev$Signl **Signal an event occurrence** **Ev$Signl**

INPUT: d0.l = event ID number
d1.w = MS bit set to activate all processes in range
LS bits = 8 (Ev$Signl function code)

OUTPUT: None

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION: The Ev$Signl function signals that an event has occurred. The current event variable is first updated with the signal auto-increment (specified when the event was created). Then the event queue is searched for the first process waiting for that event value. If the MS bit of d1.w (the function code) is set, all processes in the event queue that have a value in range are activated. The sequence is the same for each event in the queue until the queue is exhausted:

1. The signal auto-increment is added to the event variable.

2. The first process in range is awakened.

3. The event variable is updated with the wait auto-increment.

4. The search is continued with the updated value.

Ev$Pulse **Signal an event occurrence** **Ev$Pulse**

INPUT:
```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = 9 (Ev$Pulse function code)
d2.l = event pulse value
```

OUTPUT: None

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: `E$EvntID` - the ID specified is not a valid active event

FUNCTION: The Ev$Pulse function signals an event occurrence, but differs from Ev$Signl. The event variable is set to the value passed in d2.l, and the signal auto-increment is not applied. The Ev$Signl search routine is then executed and the original event value is restored.

Ev$Set **Set event variable and signal an event occurrence** **Ev$Set**

INPUT:
```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = A (Ev$Set function code)
d2.l = new event value
```

OUTPUT: None

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: `E$EvntID` - the ID specified is not a valid active event

FUNCTION: The Ev$Set function differs only slightly from the Ev$Signl call. The event variable is initially set to the value passed in d2.l instead of being updated with the signal auto-increment. After this is done, the Ev$Signal routine is executed directly.

Ev$SetR **Set relative event variable and signal an event occurrence** **Ev$SetR**

INPUT:
```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = B (Ev$SetR function code)
d2.l = (signed) increment for event variable
```

OUTPUT: None

ERROR OUTPUT:
```
cc   = carry bit set
d1.w = error code if error
```

POSSIBLE ERRORS: `E$EvntID - the ID specified is not a valid active event`

FUNCTION: The Ev$SetR function is a minor variation of Ev$Signl. Instead of using the signal auto-increment value to update the event variable, the value in d2.l is used. Arithmetic underflows or overflows are set to $80000000 or $7fffffff, respectively.

End of Chapter 15

NOTES

SECTION 4 - SYSTEM CALL DESCRIPTIONS

- User State System Calls
- I/O System Calls
- System State System Calls

INTRODUCTION TO THE SYSTEM CALL DESCRIPTIONS

OS-9/68000 SYSTEM CALL DESCRIPTIONS

System calls are used to communicate between the OS-9 operating system and assembly language level programs. There are three general categories:

1. User state system calls
2. I/O system calls
3. System state system calls

All system calls have a mnemonic name for easy reference. User and system state functions start with "F$". I/O related functions begin with "I$". The mnemonic names are defined in the relocatable library file "usr.l". This file should be linked with your programs.

The OS-9 I/O system calls are simpler to use than in many other operating systems. This is due to the fact that the calling program does not have to allocate and set up "file control blocks", "sector buffers", etc. Instead, OS-9 will return a path number word when a file/device is opened. This path number may be used in subsequent I/O requests to identify the file/device until the path is closed. OS-9 internally allocates and maintains its own data structures, and users never have to deal with them.

System state system calls are privileged and may be executed only while OS-9 is in system state (when it is processing another service request, executing a file manager, device driver, etc.). System state functions are included in this manual primarily for the benefit of those programmers who will be writing device drivers and other system-level applications. For a full description of system state and its uses, refer to Chapter 3.

The system calls are performed by loading the MPU registers with the appropriate parameters and executing a Trap #0 instruction immediately followed by a constant word (the request code). Function results (if any) will be returned in the MPU registers after OS-9 has processed the service request. A standard convention for reporting errors is used in all system calls; if an error occurred, the Carry bit of the condition code register will be set and register d1.w will contain an appropriate error code, permitting a BCS or BCC instruction immediately following the system call to branch on error/no error.

Here is an example system call for the "CLOSE" service request:

```
MOVE.W  Pathnum (a6),d0
TRAP    #0
DC.W    I$Close
BCS.S   Error
```

Using the assembler's "OS9" directive simplifies the call:

```
MOVE.W  Pathnum (a6),d0
OS9     I$Close
BCS.S   Error
```

Some system calls generate errors themselves; these are listed as **POSSIBLE ERRORS.** If the returned error code does not match any of the given possible errors, then it was probably returned by another system call made by the main call.

The **CROSS REFERENCE** listing for each service request shows related service requests and/or chapters that may yield more information.

In the service request descriptions which follow, registers not explicitly specified as input or output parameters are not altered. Strings passed as parameters are normally terminated by a null byte.

F$AllBit **Set bits in an allocation bit map** **F$AllBit**

ASSEMBLER CALL: OS9 F$AllBit

INPUT: d0.w = Bit number of first bit to set
d1.w = Bit count (number of bits to set)
(a0) = Base address of an allocation bit map

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$AllBit sets bits in the allocation map that were found by F$SchBit, and now allocated. Bit numbers range from 0 to N-1 (N is the number of bits in the allocation bit map).

In some applications it is necessary to allocate and deallocate segments of a fixed resource (such as memory). One convenient way is to set up a map that describes which blocks are available or in use. Each bit in the map represents one block. If the bit is set, the block is in use. If the bit is clear, the block is available. The F$SchBit, F$AllBit and F$DelBit system calls perform the elementary bitmap operations of finding a free segment, allocating it and returning it when it is no longer needed.

RBF uses these routines to manage cluster allocation on disks. They have been made accessible to users because they may occasionally be useful.

CROSS REFERENCE: See F$SchBit and F$DelBit.

F$Chain **Load and execute a new primary module** **F$Chain**

ASSEMBLER CALL: OS9 F$Chain

INPUT: d0.w = desired module type/language (must be program/object or 0=any)
d1.l = additional memory size
d2.l = parameter size
d3.w = number of I/O paths to copy
d4.w = priority
(a0) = module name ptr
(a1) = parameter ptr

OUTPUT: None: F$Chain does not return to the calling process.

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$NEMod

FUNCTION: F$Chain is used when it is necessary to execute an entirely new program, but without the overhead of creating a new process. It is functionally similar to a Fork command followed by an EXIT. F$Chain effectively "resets" the calling process' program and data memory areas and begins execution of a new primary module. Open paths are not closed or otherwise affected.

The sequence of operations taken by Chain is as follows:

1. The process' old primary module is unlinked.

2. The system parses the name string of the new process' "primary module" (the program that will be executed). Next, the system module directory is searched to see if a module with the same name and type/language is already in memory. If so, the module is linked. If not, the name string is used as the pathlist of a file which is to be loaded into memory. The first module in this file is linked.

3. The data memory area is reconfigured to the specified size in the new primary module's header.

4. Intercepts and any pending signals are erased.

F$CCtl Cache Control F$CCtl

ASSEMBLER CALL: OS9 F$CCtl

INPUT: d0.l = reserved, must be zero

OUTPUT: none

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

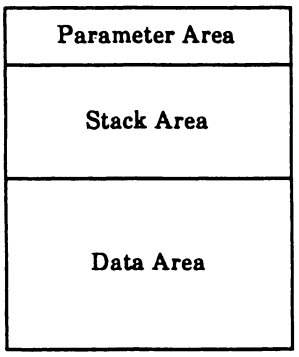
FUNCTION: F$CCtl is used to clear the hardware instruction cache, if there is one. This call must be used by any program that builds or changes executable code in memory, prior to executing it. This is necessary because the hardware instruction cache is not updated by data (write) accesses and may therefore contain the unchanged instruction(s).

For example, if a subroutine builds an OS9 system call instruction on its stack, the F$CCtl system call must be executed prior to executing the temporary instruction.

NOTE: The 68020 processor always has an instruction cache.

F$Chain **Load and execute a new primary module (continued)** **F$Chain**

The diagram below shows how Chain sets up the data memory area and registers for the new module (these are identical to F$Fork).



←(a1) (highest address)

←(a5), (a7)

←(a6) (lowest address)

Registers passed to child process:

```
sr = 0000                              (a0) = undefined
pc = module entry point                (a1) = top of memory pointer
d0.w = process ID                      (a2) = undefined
d1.l = group/user number               (a3) = primary (forked) module pointer
d2.w = priority                        (a4) = undefined
d3.w = number of I/O paths inherited   (a5) = parameter pointer
d4.l = undefined                       (a6) = static storage (data area) base
d5.l = parameter size                         pointer
d6.l = total initial memory allocation (a7) = stack pointer (same as a5)
d7.l = undefined
```

NOTE: (a6) will actually be biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. It may be significant to note when debugging programs however.

The minimum overall data area size is 256 bytes. Address registers will point to even addresses.

CROSS REFERENCE: See F$Fork and F$Load.

CAVEATS: Most errors that occur during the Chain will be returned as an exit status to the parent of the process doing the chain.

F$CmpNam **Compare two names** **F$CmpNam**

ASSEMBLER CALL: OS9 F$CmpNam

INPUT: d1.w = Length of pattern string
(a0) = Pointer to pattern string
(a1) = Pointer to target string

OUTPUT: cc = Carry bit clear if the strings match

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code if unequal or error

POSSIBLE ERRORS: E$Differ: the names do not match
E$StkOvf: the pattern is too complex

FUNCTION: F$CmpNam compares a target name to a source pattern to determine if they are equal. Upper and lower case are considered to match. Two wild card characters are recognized in the pattern string: "?" matches any single character, and "*" matches any string. The target name must be terminated by a null byte.

F$CpyMem **Copy external memory** **F$CpyMem**

ASSEMBLER CALL: OS9 F$CpyMem

INPUT:
```
d0.w = process ID of external memory's owner
d1.l = number of bytes to copy
(a0) = address of memory in external process to copy
(a1) = caller's destination buffer pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: F$CpyMem copies external memory into the user's buffer for inspection. F$CpyMem can be used to copy portions of the system's address space. This is especially helpful in examining modules. Any memory in the system may be viewed in this way.

CROSS REFERENCE: See F$Move.

F$CRC Generate CRC F$CRC

ASSEMBLER CALL: OS9 F$CRC

INPUT: d0.l = Data byte count
d1.l = CRC accumulator
(a0) = Pointer to data

OUTPUT: d1.l = Updated CRC accumulator

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$CRC generates or checks the CRC (cyclic redundancy check) values of sections of memory. F$CRC is used by compilers, assemblers or other module generators to generate a valid module CRC.

If the CRC of a new module is to be generated, the CRC is accumulated over the module (excluding CRC). The accumulated CRC is complemented and then stored in the correct position in the module.

The CRC is calculated starting at the source address over a specified number of bytes. It is not necessary to cover an entire module in one call, since the CRC may be "accumulated" over several calls. The CRC accumulator must be initialized to $FFFFFFFF before the first F$CRC call for any particular module.

When checking an existing module's CRC, the calculation should be performed on the entire module (including the module CRC). The CRC accumulator will contain the CRC constant bytes if the module CRC is correct. The CRC constant is defined in "sys.l" and "usr.l" as "CRCCon". It has the value of $00800FE3.

CAVEATS: The CRC value is three bytes long, in a four-byte field. To generate a valid module CRC, the caller must include the byte preceding the CRC in the check. This byte must be initialized to zero. For convenience, if a data pointer of zero is passed, the CRC will be updated with one zero data byte. F$CRC will always return $FF in the most significant byte of d1, so d1.l may be directly stored (after complement) in the last four bytes of a module as the correct CRC.

CROSS REFERENCE: See Chapter 1 section on CRC.

F$DatMod **Create data module** **F$DatMod**

ASSEMBLER CALL: OS9 F$DatMod

INPUT: d0.l = size of data required (not including header or CRC)
d1.w = module attr/revision
d2.w = module access permission
(a0) = module name string ptr

OUTPUT: d0.w = module type/language
d1.w = module attr/revision
(a0) = updated name string ptr
(a1) = module data ptr ('execution' entry)
(a2) = module header ptr

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BNam, E$KwnMod

FUNCTION: F$DatMod creates a data module with the specified attribute/revision and clears the data portion of the module. The module is initially created with a valid CRC, and entered into the system module directory. Several processes can communicate with each other using a shared data module.

Care must be taken not to alter the data module's header or name string to avoid the possibility of the module becoming unknown to the system.

CROSS REFERENCE: See F$SetCRC and F$Move.

CAVEATS: The module created will contain at least d0.l usable data bytes, but may be somewhat larger. The module itself will be larger by at least the size of the module header and CRC, and rounded up to the nearest system memory allocation boundary.

F$DelBit **Deallocate in a bit map** **F$DelBit**

ASSEMBLER CALL: OS9 F$DelBit

INPUT: d0.w = Bit number of first bit to clear
d1.w = Bit count (number of bits to clear)
(a0) = Base address of an allocation bit map

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: DelBit clears bits in the allocation bit map that were previously allocated and are now free for general use. Bit numbers range from 0 to N-1 (N is the number of bits in the allocation bit map).

CROSS REFERENCE: See F$AllBit and F$SchBit.

F$DExec **Execute debugged program** **F$DExec**

ASSEMBLER CALL: OS9 F$DExec

```
INPUT:  d0.w = process ID of child to execute
        d1.l = number of instructions to execute (0 = continuous)
        d2.w = number of breakpoints in list
        (a0) = breakpoint list
        register buffer contains child register image

OUTPUT: d0.l = total number of instructions executed so far
        d1.l = remaining count not executed
        d2.w = exception occurred, if non-zero; exception offset
        d3.w = classification word  (addr or bus trap only)
        d4.l = access address       (addr or bus trap only)
        d5.w = instruction register (addr or bus trap only)
        register buffer updated

ERROR OUTPUT:   cc = Carry bit set
                d1.w = Appropriate error code
```

POSSIBLE ERRORS: E$IPrcID, E$PrcAbt

FUNCTION: F$DExec controls the execution of a suspended child process that has been created by the F$DFork call. The process performing F$DExec is suspended and its debugged child process is executed instead. Once the specified number of instructions have been executed, a breakpoint is reached or an unexpected exception occurs, execution terminates and control is returned to the parent process. Consequently, the parent and the child processes are never active at the same time.

F$DExec works by tracing every instruction of the child process. It checks for the termination conditions after each instruction. Breakpoints are simply lists of addresses to check and work with ROMed object programs. Consequently, the child process being debugged run at a slow speed.

If a "-1" (hex $FFFFFFFF) is passed in d0.l, F$DExec will replace the instruction at each breakpoint address with an illegal opcode. It then executes the child process at full speed (with the trace bit clear), until a breakpoint is reached or the program terminates. This can save an enourmous amount of time. When this method is used, however, it is impossible for F$DExec to count the number of executed instructions.

Any OS-9 system calls made by the suspended program are executed at "full speed" and are considered one logical instruction. The same is true of system state trap handlers. System state processes can not be debugged.

The register buffer passed in the F$DFork call is used by the system to save and restore the child's registers. Changing the contents of the register buffer will alter the child process' registers.

If the child process terminates for any reason, the carry bit is set and returned. Tracing may continue as long as the child process does not perform an F$Exit (even after encountering any normally fatal error). An F$DExit call must be made for the debugged process' resources (memory) to be returned.

CROSS REFERENCE: See F$DFork and F$DExit.

CAVEATS: Tracing is allowed through user state trap handlers, intercept routines and the F$Chain system call. This is not a problem, but may seem strange at times.

F$DExit **Exit debugged program** **F$DExit**

ASSEMBLER CALL: OS9 F$DExit

INPUT: d0.w = process ID of child to terminate

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$IPrcID

FUNCTION: F$DExit terminates a suspended child process that was created with the F$DFork system call. Normal termination by the child process does not release any of its resources, in order to permit post-mortem examination.

CROSS REFERENCE: See F$Exit, F$DFork and F$DExec.

F$DFork **Fork process under control of debugger** **F$DFork**

ASSEMBLER CALL: OS9 F$DFork

INPUT:
```
d0.w = desired module type/revision (0 = any)
d1.l = additional stack space to allocate (if any)
d2.l = parameter size
d3.w = number of I/O paths for child to inherit
d4.w = module priority
(a0) = module name ptr (or pathlist)
(a1) = parameter ptr
(a2) = register buffer: copy of child's (d0-d7/a0-a7/sr/pc)
```

OUTPUT:
```
d0.w = child process ID
(a0) = updated past module name string
(a2) = initial image of the child process' registers in buffer
```

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: F$DFork works similar to F$Fork, except that it is provided for a debugger utility to create a process whose execution can be closely controlled. The process created is not placed in the active queue. Instead, it is left in a "suspended" state, allowing the debugger to control its execution. This is done through the special system calls: F$DExec and F$DExit.

The child process is created with the trace bit of its status register set. It is executed with the F$DExec system call.

The register buffer is an area in the caller's data area that is permanently associated with each child process. It will be set to an image of the child's initial registers for use with the F$DExec call.

For information about process creation, please see the F$Fork service request definition.

CROSS REFERENCE: See F$Fork, F$DExec and F$DExit.

CAVEATS: A process created by the F$DFork will never execute at all unless it is told to do so. When it is run, the trace bit will be set in the user status register causing the system trace exception handler to occur once for each user instruction executed. This makes user programs run slow.

Processes whose primary module is owned by a super-user may only be debugged by a super-user. System state processes may not be debugged.

F$Exit **Terminate the calling process** **F$Exit**

ASSEMBLER CALL: OS9 F$Exit

INPUT: `d1.w = Status code to be returned to the parent process`

OUTPUT: Process is terminated

ERROR OUTPUT: None

FUNCTION: F$Exit is the means by which a process can terminate itself. Its data memory area is deallocated, and its primary module is UnLinked. All open paths are automatically closed.

The death of the process can be detected by the parent executing a F$Wait call. This returns (to the parent) the status word passed by the child in its Exit call. The status word can be an OS-9 error code that the terminating process wishes to pass back to its parent process (the shell assumes this). It could also be used to pass a user-defined status value.

Processes to be called directly by the shell should only return an OS-9 error code or zero if no error occurred. Note that the parent MUST do an F$Wait before the process descriptor is returned.

The following information describes the operation of an F$Exit call.

1. Close all paths.
2. Return memory to system.
3. Unlink primary module and user trap handlers.
4. Free process descriptor of any dead child processes.
5. If parent is dead, free the process descriptor.
6. If parent has not executed an F$Wait call, leave the process in limbo until parent notices the death.
7. If parent is is waiting, move parent to active queue, inform parent of death/status, remove child from sibling list and free its process descriptor memory.

CROSS REFERENCE: See I$Close, F$SrtMem, F$UnLink, F$FindPD, F$RetPD, F$Fork, F$Wait and F$AProc.

CAVEATS: Only the primary module and the user trap handlers are unlinked. Any other modules that are loaded or linked by the process should be unlinked before calling F$Exit.

Although F$Exit closes any open paths, it pays no attention to errors returned by the F$Close request. Because of I/O buffering, this can cause write errors to go unnoticed, when paths ARE left open. However, by convention, the standard I/O paths (0,1,2) are usually left open.

F$Fork **Create a new process** **F$Fork**

ASSEMBLER CALL: OS9 F$Fork

INPUT:
```
d0.w = desired module type/revision (usually program/object 0=any)
d1.l = additional memory size
d2.l = parameter size
d3.w = number of I/O paths to copy
d4.w = priority
(a0) = module name pointer
(a1) = parameter pointer
```

OUTPUT:
```
d0.w=child process ID
(a0)=updated beyond module name
```

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

POSSIBLE ERRORS: E$NEMod

FUNCTION: F$Fork creates a new process which becomes a "child" of the caller. It sets up the new process' memory, MPU registers and standard I/O paths.

The system parses the name string of the new process' "primary module" (the program that will initially be executed). Next, the system module directory is searched to see if the program is already in memory. If so, the module is linked and executed. If not, the name string is used as the pathlist of the file which is to be loaded into memory. The first module in this file is linked and executed. To be loaded, the module must be program object code and have the appropriate read and/or execute permissions set for the user.

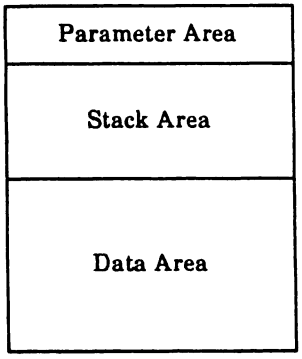
The primary module's module header is used to determine the process' initial data area size. OS-9 then attempts to allocate RAM equal to the required data storage size plus any additional size given in d1, plus the size of any parameter passed. The RAM area must be contiguous.

The new process' registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point. If d4.w is set to zero, the new process will inherit the same priority as the calling process.

When the shell processes a command line, it passes a copy of the parameter portion (if any)of the command line as a parameter string. The shell appends an end-of- line character to the parameter string to simplify string-oriented processing.

If any of the these operations are unsuccessful, the Fork is aborted and the caller is returned an error.

The diagram below shows how Fork sets up the data memory area and registers for a newly-created process. See F$Wait.



←(a1) (highest address)

←(a5), (a7)

←(a6) (lowest address)

Registers passed to child process:

```
sr = 0000                                  (a0) = undefined
pc = module entry point                    (a1) = top of memory pointer
d0.w = process ID                          (a2) = undefined
d1.l = group/user number                   (a3) = primary (forked) module pointer
d2.w = priority                            (a4) = undefined
d3.w = number of I/O paths inherited       (a5) = parameter pointer
d4.l = undefined                           (a6) = static storage (data area) base
d5.l = parameter size                             pointer
d6.l = total initial memory allocation     (a7) = stack pointer (same as a5)
d7.l = undefined
```

NOTE: (a6) will actually be biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. It may be significant to note when debugging programs however.

CROSS REFERENCE: See F$Wait, F$Exit and F$Chain.

CAVEATS: Both the child and parent process will execute concurrently. If the parent executes an F$Wait call immediately after the fork, it will wait until the child dies before it resumes execution. A child process descriptor is returned only when the parent does an F$Wait call.

Modules owned by a super-user can be made to execute in system state by setting the system state bit in the module's attributes. **This is rarely necessary, quite dangerous and not recommended for casual users.**

F$GBlkMp **Get Free Memory Block Map** **F$GBlkMp**

ASSEMBLER CALL: OS9 F$GblkMp

INPUT: d0.l = Address to begin reporting segments
d1.l = Size of buffer in bytes
(a0) = Buffer pointer

OUTPUT: d0.l = System's minimum memory allocation size
d1.l = Number of memory fragments in system
d2.l = Total RAM found by system at startup
d3.l = Current total free RAM available
(a0) = Memory fragment information

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$GBlkMp copies the address and size of the system's free RAM blocks into the user's buffer for inspection. It also returns various information concerning the free RAM as noted by the output registers above. The address and size of the free RAM blocks are returned in the user's buffer in following format (address and size are 4-bytes):

address	size
address	size
address	size

.
.
.

address	size
0	

end of memory → fragment information

Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be used accessed directly. Use F$SRqMem to request free memory blocks.

CROSS REFERENCE: See F$SRqMem and F$Mem.

CAVEATS: This system call provides a status report concerning free system memory for "mfree" and similar utilities. The address and size of free RAM changes with system use. Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be used accessed directly. Use F$SRqMem to request free memory blocks.

F$GModDr **Get module directory** **F$GModDr**

ASSEMBLER CALL: OS9 F$GModDr

INPUT: d1.l = Maximum number of bytes to copy
(a0) = Buffer pointer

OUTPUT: d1.l = Actual number of bytes copied

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$GModDr copies the system's module directory into the user's buffer for inspection. F$GModDr is used by Mdir to look at the module directory. Although the module directory contains pointers to each module in the system, the modules should never be accessed directly. Rather, an F$CpyMem call should be used to copy portions of the system's address space for inspection. On some systems, directly accessing the modules may cause address or bus trap errors.

CROSS REFERENCE: See F$Move and F$CpyMem.

CAVEATS: This system call is provided primarily for use by "mdir" and similar utilities. The format and contents of the module directory may change on different releases of OS-9. For this reason, it is often preferable to use the output of "mdir" to determine the names of modules in memory.

F$GPrDBT Get process descriptor block table copy F$GPrDBT

ASSEMBLER CALL: OS9 F$GPrDBT

INPUT: d1.l = maximum number of bytes to copy
(a0) = Buffer pointer

OUTPUT: d1.l = Actual number of bytes copied

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$GPrDBT copies the process descriptor block table into the caller's buffer for inspection. F$GPrDBT is used by the Procs utility to determine quickly which processes are active in the system. Although F$GPrDBT returns pointers to the process descriptors of all processes, the process descriptors should NEVER be accessed directly. Instead, the F$GPrDsc system call should be used if it is necessary to inspect particular process descriptors.

The system call, F$AllPd, describes the format of the process descriptor block table.

CROSS REFERENCE: See F$GPrDsc and F$AllPD.

F$GPrDsc **Get process descriptor copy** **F$GPrDsc**

ASSEMBLER CALL: OS9 F$GPrDsc

INPUT:
```
d0.w = Requested process ID
d1.w = Number of bytes to copy
(a0) = Process descriptor buffer pointer
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

POSSIBLE ERRORS: E$PrcID

FUNCTION: F$GPrDsc copies a process descriptor into the caller's buffer for inspection. There is no way to change data in a process descriptor. F$GPrDsc is used by the Procs utility to gain information about an existing process.

CROSS REFERENCE: See F$GPrDBT.

CAVEATS: The format and contents of a process descriptor may change with different releases of OS-9.

F$Gregor **Get gregorian date** **F$Gregor**

ASSEMBLER CALL: OS9 F$Gregor

INPUT: d0.l = time (seconds since midnight)
d1.l = Julian date

OUTPUT: d0.l = time (00hhmmss)
d1.l = date (yyyymmdd)

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$Gregorian converts julian dates to gregorian dates. Gregorian dates are what is considered the "normal" calendar dates.

The julian date is similar to the julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

CAVEATS: The normal (Gregorian) calendar was revised to correct errors due to leap year at different dates throughout the world. The algorithm used by OS-9 makes this adjustment on October 15, 1582. Care must be taken when working with old dates, because the same day may be recorded as a different date by different sources.

NOTE: F$Gregor is the inverse function of F$Julian.

CROSS REFERENCE: F$Julian and F$Time.

F$ID Get process ID / user ID F$ID

ASSEMBLER CALL: OS9 F$ID

INPUT: None

OUTPUT: d0.w = Current process ID
d1.l = Current process group/user number
d2.w = Current process priority

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: Returns the caller's process ID number, group and user ID, and current process priority (all word values). The process ID is assigned by OS-9 and is unique to the process. The user ID is defined in the system password file, and is used for system and file security. Several processes can have the same user ID.

F$Icpt **Set up a signal intercept trap** **F$Icpt**

ASSEMBLER CALL: OS9 F$Icpt

INPUT: (a0) = Address of the intercept routine
(a6) = Address to be passed to the intercept routine

OUTPUT: Signals sent to the process will cause the intercept routine to be called instead of the process being killed.

ERROR OUTPUT: None

FUNCTION: F$Icpt tells OS-9 to install a signal intercept routine where (a0) contains the address of the signal handler routine, and (a6) usually contains the address of the program's data area.

After the F$Icpt call has been made, whenever the process receives a signal, its intercept routine will be executed. A signal will abort a process which has not used the F$Icpt service request and its termination status (d1.w register) will be the signal code. Many interactive programs set up an intercept routine to handle keyboard abort and keyboard interrupt signals.

The intercept routine is entered asynchronously because a signal may be sent at any time (similar to an interrupt) and is passed the following:

```
d1.w = Signal code
(a6) = Address of intercept routine data area
```

The intercept routine should be short and fast, such as setting a flag in the process' data area. Complicated systen calls (such as I/O) should be avoided. After the intercept routine has been completed, it may return to normal process execution by executing the F$RTE system call.

CROSS REFERENCE: See F$RTE and F$Send.

CAVEATS: Each time the intercept routine is called, 70 bytes are used on the user's stack.

F$Julian **Get julian date** **F$Julian**

ASSEMBLER CALL: OS9 F$Julian

INPUT: d0.l = time (00hhmmss)
d1.l = date (yyyymmdd)

OUTPUT: d0.l = time (seconds since midnight)
d1.l = Julian date

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$Julian converts gregorian dates to julian dates.

Julian dates are very convenient for computing elapsed time. To compute the number of days between two dates, the julian day numbers may be directly subtracted.

The julian day number returned is similar to the julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

The Julian day number may also be used to determine the day of the week for a given date. Use the following formula:

```
weekday = MOD(Julian_Date + 2, 7)
```

This will return the day of the week as 0 = Sunday, 1 = Monday, etc.

CAVEATS: The normal (Gregorian) calendar was revised to correct errors due to leap year at different dates throughout the world. The algorithm used by OS-9 makes this adjustment on October 15, 1582. Care must be taken when working with old dates, because the same day may be recorded as a different date by different sources.

F$Link **Link to memory module** **F$Link**

ASSEMBLER CALL: OS9 F$Link

INPUT: d0.w = Desired module type/language byte (0 = any)
(a0) = Module name string pointer

OUTPUT: d0.w = Actual module type/language
d1.w = Module attributes/revision level
(a0) = Updated past the module name
(a1) = Module execution entry point
(a2) = Module pointer

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: EMNF, EBNam, E$ModBsy

FUNCTION: F$Link causes OS-9 to search the module directory for a module having a name, language, and type as given in the parameters. If found, the address of the module's header is returned in (a2). The absolute address of the module's execution entry point is returned in (a1) (as a convenience; this and other information can be obtained from the module header). The module's "link count" is incremented to keep track of how many processes are using the module. If the module requested is not re-entrant, only one process may link to it at a time.

If the module's access word does not give the process read permission, the link call will fail. Link also fails to find modules whose header has been destroyed (altered or currupted) in memory.

CROSS REFERENCE: See F$Load, F$UnLink and F$UnLoad.

F$Load **Load module(s) from a file** **F$Load**

ASSEMBLER CALL: OS9 F$Load

INPUT: d0.b = Access mode
(a0) = Path name pointer

OUTPUT: d0.w = Actual module type/language
d1.w = Attributes/revision level
(a0) = Updated beyond path name
(a1) = Module execution entry pointer (of first module loaded)
(a2) = Module pointer

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$MemFul, E$BMID

FUNCTION: F$Load opens a file specified by the pathlist. It reads one or more memory modules from the file into memory until an error or end of file is reached. Then it closes the file. Modules are usually loaded into the highest physical memory available.

An error can be an actual I/O error, a module with a bad parity or CRC or the system memory is full.

All modules that are loaded are added to the system module directory, and the first module read is Linked. The parameters returned are the same as the Link call and apply only to the first module loaded.

In order to be loaded, the file must contain a module or modules that have a proper module header and CRC. The access mode may be specified as either Exec__ or Read__, causing the file to be loaded from the current execution or data directory, respectively.

If any of the modules loaded belong to the super-user, the file must also be owned by the super-user. This is a protection that prevents normal users form executing privileged service requests.

CAVEATS: F$Load will not work on SCF devices.

F$Mem **Resize data memory area** **F$Mem**

ASSEMBLER CALL: OS9 F$Mem

INPUT: d0.l = Desired new memory size in bytes

OUTPUT: d0.l = Actual size of new memory area in bytes
(a1) = Pointer to new end of data segment (+1)

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$DelSP, E$MemFul, E$NoRAM

FUNCTION: F$Mem is used to contract or expand the process' data memory area. The new size requested is rounded up to an even memory allocation block (16 bytes in version 2.0). Additional memory is allocated contiguously upward (towards higher addresses), or deallocated downward from the old highest address. If d0 equals zero, the call is taken to be an information request and the current upper bound and size will be returned.

This request can never return all of a process' memory, or cause the memory at its current stack pointer to be deallocated.

The request may return an error upon an expansion request even though adequate free memory exits because the data area must always be contiguous, and memory requests by other processes may fragment memory into smaller, scattered blocks that are not adjacent to the caller's present data area.

F$PErr **Print error message** **F$PErr**

ASSEMBLER CALL: OS9 F$PErr

INPUT: d0.w = Error message path number (0=none)
d1.w = Error number

OUTPUT: None

ERROR OUTPUT: None

FUNCTION: F$PErr is the system's error reporting facility. It writes an error message to the standard error path. Most OS-9 systems will print "ERROR #mmm.nnn". Error numbers 000:000 to 063:255 are reserved for the operating system.

If an error path number is given the path is searched for a text description of the error encountered. The error message path contains an ASCII file of error messages. Each line may be up to 80 characters long. If the error number matches the first seven characters in a line (i.e. "000:215"), the rest of the line will be printed along with the error number.

Error messages may be continued on several lines by beginning each continuation line with a space. An example error file might contain lines like this:

```
000:214 (E$FNA) File not accessible.
  An attempt to open a file failed.  The file was found, but is
  inaccessible to you in the requested mode.  Check the file's owner
  ID and access attributes.

000:215 (E$BPNam) Bad pathlist specified.
  The pathlist specified is syntactically incorrect.

000:216 (E$PNNF) File not found.
  The pathlist does not lead to any known file.

000:218 (E$CEF) Tried to create a file that already exists.

000:253 (E$Share) Non-sharable file busy.
  The most common way to get this error is to try and delete a file
  that is currently open.  Any time a file is opened for
  non-sharable access, but already in use this error will occur.
  This error also occurs if you try to access a non-sharable device
  (such as a printer) that is busy.
```

F$PrsNam **Parse a path name** **F$PrsNam**

ASSEMBLER CALL: OS9 F$PrsNam

INPUT: (a0) = Name of string pointer

OUTPUT: d0.b = Pathlist delimiter
d1.w = Length of pathlist element
(a0) = Pathlist pointer updated past the optional "/" character
(a1) = Address of the last character of the name +1

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BNam

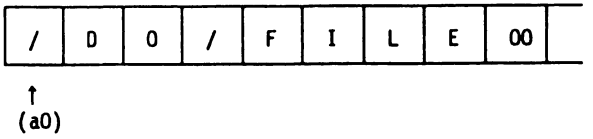
FUNCTION: F$PrsNam parses a string for a valid pathlist element, returning its size. Note that this does not parse an entire pathname, only one element in it. A valid pathlist element may contain the following characters:

A - Z	Upper case letters	.	Periods
a - z	Lower case letters	_	Underscores
0 - 9	Numbers	$	Dollar signs

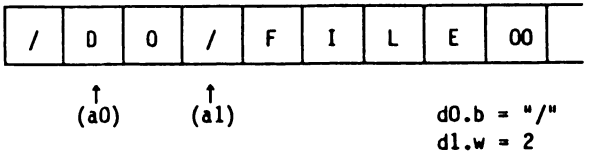
Any other character will terminate the name and be returned as the pathlist delimiter.

NOTE: PrsNam processes only one name, so several calls may be needed to process a pathlist that has more than one name. F$PrsNam will terminate a name on recognizing a delimiter character. Pathlists must usually be terminated with a null byte.

BEFORE F$PrsNam CALL:



AFTER F$PrsNam CALL:



F$RTE **Return from interrupt exception** **F$RTE**

ASSEMBLER CALL: OS9 F$RTE

INPUT: None

OUTPUT: None

FUNCTION: F$RTE may be used to exit from a signal processing routine.

F$RTE is used to terminate a process signal intercept routine and continue execution of the main program. However, if there are unprocessed signals pending, the interrupt routine will be executed again (until the queue is exhausted) before returning to the main program.

CAVEATS: When a signal is received 70 bytes are used on the user stack. Consequently intercept routines should be kept very short and fast, if many signals are expected.

CROSS REFERENCE: See F$Icpt.

F$SchBit **Search bit map for a free area** **F$SchBit**

ASSEMBLER CALL: OS9 F$SchBit

INPUT:
```
d0.w = Beginning bit number to search
d1.w = Number of bits needed
(a0) = Bit map pointer
(a1) = End of bit map (+1) pointer
```

OUTPUT:
```
d0.w = Beginning bit number found
d1.w = Number of bits found
```

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: F$SchBit searches the specified allocation bit map for a free block (cleared bits) of the required length, starting at the beginning bit number (d0.w). SchBit returns the offset of the first block found of the specified length.

If no block of the specified size exists, it returns with the carry set, beginning bit number, and size of the largest block found.

CROSS REFERENCE: See F$AllBit and F$DelBit.

F$Send **Send a signal to another process** **F$Send**

ASSEMBLER CALL: OS9 F$Send

INPUT: d0.w = Intended receiver's process ID number (0 = all)
d1.w = Signal code to send

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$IPrcID, E$USigP

FUNCTION: F$Send sends a signal to a specific process. The signal code is a word value. A process may send the same signal to multiple processes of the same Group/User ID by passing 0 as the receiver's process ID number. For example, the OS-9 Shell command, "kill 0", will unconditionally abort all processes with the same group/user ID (except the Shell itself). This is a handy but dangerous tool to get rid of unwanted background tasks. If an attempt is made to send a signal to a process that has an unprocessed, previous signal pending, the signal is placed in a first-in, first-out (FIFO) queue of signals for the indivdual process. If the process is in the signal intercept routine when it receives a signal, the new signal is processed when F$RTE is executed.

If the destination process for the signal is sleeping or waiting, it will be activated so that it may process the signal. The signal processing intercept routine will be executed, if it exists (see F$Icpt), otherwise the signal will abort the destination process, and the signal code becomes the exit status (see F$Wait).

An exception is the Wakeup signal. It activates a sleeping process but does not cause the signal intercept routine to be examined and will not abort a process that has not made an F$Icpt call.

Some of the signal codes have meanings defined by convention:

```
S$Kill   = 0 = System abort (unconditional)
S$Wake   = 1 = Wake'up process
S$Abort  = 2 = Keyboard abort
S$Intrpt = 3 = Keyboard interrupt
256-65535 = User defined
```

If an attempt is made to send a signal to a process that has an unprocessed, previous signal pending, the current send request will be ignored and an error will be returned. An attempt can be made to try and send the signal later. It is good practice to issue a sleep call for a few ticks before a retry to avoid wasting CPU time.

The S$Kill signal may only be sent to processes with the same group ID as the sender. Super users may kill any process.

CROSS REFERENCE: See F$Wait, F$Icpt and F$Sleep.

F$SetCRC **Generate valid CRC in module** **F$SetCRC**

ASSEMBLER CALL: OS9 F$SetCRC

INPUT: (a0) = module pointer

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BMID

FUNCTION: F$SetCRC updates the header parity and CRC of a module in memory. The module may be an existing module known to the system, or simply an image of a module that will subsequently be written to a file. The module must have correct size and sync bytes; other parts of the module are not checked.

CROSS REFERENCE: See F$CRC.

CAVEATS: The module image must start on an even address or an address error will occur.

F$SetSys **Set/Examine OS-9 system global variables** **F$SetSys**

ASSEMBLER CALL: OS9 F$SetSys

INPUT: d0.w = offset of system global variable to set/examine
d1.l = size of variable in least significant word (1, 2 or 4 bytes).
The most significant bit, if set, indicates an examination request.
Otherwise, the variable is changed to the value in register d2.
d2.l = new value (if change request)

OUTPUT: d2.l = original value of system global variable.

ERROR OUTPUT: cc = Carry set
d1.w = Appropriate error code

FUNCTION: This function call is used to change or examine a system global variable. These variables have a "D_" prefix in the system library "sys.l". Consult the DEFS files for a description of the system global variables.

CROSS REFERENCE: See F$SPrior and Chapter 12 on using the DEFS Files.

CAVEATS: System variables may only be changed by a super user. Any system variable may be examined, but only a few may be altered. The only useful variables that may be changed are D_MinPty and D_MaxAge. Consult the section on process scheduling for an explanation of what these variables control.

The system global variables are OS-9's data area. They are highly likely to change from one release to another. Programs using this system call will probably have to be re-linked to be able to run on future versions of OS-9.

CAUTION: Be careful when changing system global variables.

F$Sleep **Put calling process to sleep** **F$Sleep**

ASSEMBLER CALL: OS9 F$Sleep

INPUT: d0.l = Ticks/seconds (length of time to sleep)

OUTPUT: d0.l = Remaining number of ticks if awakened prematurely

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$NoClk

FUNCTION: F$Sleep deactivates the calling process until the number of ticks requested have elapsed. Sleep(0) will sleep indefinitely. Sleep(1) gives up a time slice but does not necessarily sleep for one tick. Due to the fact that it is not known when the F$Sleep request was made during the current tick, F$Sleep cannot be used to time more accurately than + or - 1 tick.

A sleep of one tick is effectively a "give up current time slice" request; the process is immediately inserted into the active process queue and will resume execution when it reaches the front of the queue.

A sleep of two or more (n) ticks causes the process to be inserted into the active process queue after n - 1 ticks occur and will resume execution when it reaches the front of the queue. The process will be activated before the full time interval if a signal (in particular SS.Wake) is received. Sleeping indefinitely is a good way to wait for a signal or interrupt without wasting CPU time.

The duration of a "tick" is system dependent but is usually .01 seconds. If the high order bit of d0.l is set, the low 31 bits are converted from 256ths of a second into ticks before sleeping to allow program delays to be independent of the system's clock rate.

CROSS REFERENCE: See F$Send, F$Wait.

CAVEATS: The system clock must be running to perform a timed sleep. The system clock is not required to perform an indefinite sleep or to give up a timeslice.

F$SigMask Masks/Unmasks Signals During Critical Code F$SigMask

ASSEMBLER CALL: OS9 F$SigMask

INPUT: d0.l = reserved, must be zero
d1.l = process signal level (0=clear, 1=set/increment -1=decrement)

OUTPUT: none

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

FUNCTION: F$SigMask is used to enable or disable signals from reaching the calling process. If a signal is sent to a process whose mask is disabled, the signal will be queued until the process mask becomes enabled. The process' signal intercept routine is executed with signals inherently masked.

Two exceptions to this rule are the S$Kill and S$Wake signals. S$Kill terminates the receiving process, regardless of the state of its mask. S$Wake insures that the process is active, but does not queue.

When a process makes an F$Sleep or F$Wait system call, its signal mask is automatically cleared. If a signal is already queued, these calls return immediately (t the intercept routine).

NOTE: Signals are analagous to hardware interrupts. They should be masked sparingly, and intercept routines should be as short and fast as possible.

F$SPrior — Set process priority — F$SPrior

ASSEMBLER CALL: OS9 F$SPrior

INPUT: d0.w = Process ID number
d1.w = Desired process priority:
65535 = highest
0 = lowest

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$IPrcID

FUNCTION: F$SPrior changes the process priority to the new value given. A process can only change another process' priority if it has the same user ID. The one exception to this rule is a super user (group ID zero) which may alter any process' priority.

There are two system global variables that effect task switching. D_MinPty is the minimum priority that a task must have for OS-9 to age or execute it. D_MaxAge is the cutoff aging point. D_MinPty and D_MaxAge are initially set in the INIT module.

CROSS REFERENCE: See F$SetSys and the section on process scheduling.

CAVEATS: A very small change in relative priorities has a large effect. For example, if two processes have the priorities 100 and 200, the process with the higher priority will run 100 times before the low priority process runs at all. In actual practice, the difference may not be this extreme because programs spend a lot of time waiting for I/O devices.

F$SRqMem **System memory request** **F$SRqMem**

ASSEMBLER CALL: OS9 F$SRqMem

INPUT: d0.l = Byte count of requested memory

OUTPUT: d0.l = Byte count of memory granted
(a2) = Pointer to memory block allocated

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$MemFul, E$NoRAM

FUNCTION: F$SRqMem allocates a block of memory from the top of available RAM. The number of bytes requested will be rounded up to a system defined blocksize (currently 16 bytes). This system call is useful for allocating I/O buffers and any other semi-permanent memory. The memory will always begin on an even boundary. If -1 is passed in d0.l, the largest block of free memory is allocated to the calling process.

The maximum number of blocks any process may have allocated is 32. This includes the primary module's static storage area. Note that this is a limit on the number of segments allocated, not the amount of memory.

CROSS REFERENCE: See F$SRtMem and F$Mem.

CAVEATS: The byte count of memory allocated (as well as the pointer to the block allocated) must be saved if the memory is ever to be returned to the system.

F$SRtMem **Return system memory** **F$SRtMem**

ASSEMBLER CALL: OS9 F$SRtMem

INPUT: d0.l = Byte count of memory being returned
(a2) = Address of memory block being returned

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPAddr

FUNCTION: De-allocates memory after it is no longer needed. The number of bytes returned will be rounde up to a system defined blocksize before the memory is returned. Rounding occurs identically to that done by F$SRqMem.

In user state, the system keeps track of memroy allocated to a process and all blocks not returned are automatically de-allocated by the system when a process terminates. In system state, the process must explicitely return its memory or it is lost.

CROSS REFERENCE: See F$SRqMem and F$Mem.

F$SSpd **Suspend process** **F$SSpd**

ASSEMBLER CALL: OS9 F$SSpd

INPUT: d0.w = process ID to suspend

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$SSpd is currently not implemented.

CROSS REFERENCE: See F$SetPri and F$SetSys.

CAVEATS: A process may be suspended by setting its priority below the system's minimum executable priority level (D__SysMin).

F$STime **Set system date and time** **F$STime**

ASSEMBLER CALL: OS9 F$STime

INPUT: d0.l = current time (00hhmmss)
d1.l = current date (yyyymmdd)

OUTPUT: Time/date is set

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$STime is used to set the current system date/time and start the system real-time clock to produce time slice interrupts. F$STime is accomplished by putting the date/time packet in the system direct storage area, and then linking the clock module. The clock initialization routine is called if the link is successful.

It is the duty of the clock module to:

1. Set up any hardware dependent functions to produce system tick interrupts (including moving new date/time into hardware, if needed).

2. Install a service routine to clear the interrupt when a tick occurs.

The OS-9 kernel takes care of keeping track of the current date and time in software to make clock modules small and simple. Certain utilities and functions in OS-9 expect the clock to be running with an accurate date and time. For this reason, Setime should always be run when the system is started. This is usually done in the system startup file.

CROSS REFERENCE: See F$Link and F$Time.

CAVEATS: The date and time are not checked for validity in any way. On systems with a battery backed clock, it is usually only necessary to supply the year to the Setime call. The actual date and time will be read from the real time clock.

F$STrap **Set error trap handler** **F$STrap**

ASSEMBLER CALL: OS9 F$STrap

INPUT: (a0) = Stack to use if exception occurs (or zero to use the current stack)
(a1) = Pointer to service request initialization table

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$STrap enters "process local" Error Trap routine(s) into the process descriptor dispatch table. If an entry for a particular routine already exists, it is replaced.

The following exception errors may be caught by user programs:

```
Bus error
Address error
Illegal instruction
Zero Divide
CHK instruction
TRAPV instruction
Privilege violation
Line 1010 emulator
Line 1111 emulator
```

On 68020 systems with a 68881 floating point coprocessor, the following exception errors may also be caught

```
Branch or set on unordered condition
Inexact result
Divide by zero
Underflow
Overflow
NAN signalled
```

If a user routine is not provided and one of these exceptions occurs, the program is aborted. An example initialization table might look like:

```
ExcpTbl   dc.w    T_TRAPV,OvfError-*-4
          dc.w    T_CHK,CHKError-*-4
          dc.w    -1 End of Table
```

When an exception routine is executed, it will be passed the following:

```
d7.w = Exception vector offset
(a0) = Program counter when exception occurred (same as R$PC(a5))
(a1) = Stack pointer when exception occurred (R$a7(a5))
(a5) = User's register stack image when exception occurred
(a6) = user's primary global data pointer
```

To return to normal program execution after handling the error, the exception must restore all registers (from the register image at (a5)), and jump to the return program counter. For some kinds of exceptions (especially bus and address errors) this may not be appropriate. It is the user program's responsibility to determine whether and where to continue execution.

It is possible to disable an error exception handler. This is done by calling F$STrap with an initialization table that specifies zero as the offset to the routine(s) that are to be removed. For example, the following table would remove user routines for the TRAPV and CHK error exceptions:

```
Table   dc.w  T_TRAPV, 0
        dc.w  T_CHK, 0
        dc.w  -1
```

CAVEATS: Beware of exceptions in exception handling routines. They are usually non-reentrant.

F$SUser **Set user ID number** **F$SUser**

ASSEMBLER CALL: OS9 F$SUser

INPUT: d1.l = Desired group/user ID number

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$SUser alters the current user ID to the specified ID. The following restrictions govern the use of F$SUser:

1. User number 0.0 may change their ID to anything without restriction.
2. A primary module owned by user 0.0 may change its ID to anything without restriction.
3. Any primary module may change its user ID to match the module's owner.

All other attempts to change user ID number will return an E$Permit error.

F$SysDbg **Call system debugger** **F$SysDbg**

ASSEMBLER CALL: OS9 F$SysDbg

INPUT: None

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry set
d1.w = Appropriate error code
```

FUNCTION: F$SysDbg invokes the system level debugger, if one exists, to allow system state routines, such as device drivers, to be debugged. The system level debugger runs in system state and effectively stops timesharing whenever it is active. It should never be used when there are other users on the system. This call can be made only by a user with a group.user ID of 0.0.

CAVEATS: The system debugger must be "enabled" before installing breakpoints or attempting to trace instructions. If no system debugger is available, the system will be reset. The system debugger takes over some of the exception vectors directly, in particular the Trace exception. This makes it impossible to use the user debugger when the system debugger is enabled.

F$Time **Get system date and time** **F$Time**

ASSEMBLER CALL: OS9 F$Time

INPUT: d0.w = Format
0 = Gregorian
1 = Julian
2 = Gregorian with ticks
3 = Julian with ticks

OUTPUT: d0.l = Current time
d1.l = Current date
d2.w = day of week (0 = Sunday to 6 = Saturday)
d3.l = tick rate/current tick (if requested)

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$Time returns the current system date and time. In the (normal) gregorian format, time is expressed as 00hhmmss, and date as yyyymmdd. The julian format expresses time as seconds since midnight, and date as the julian day number. This can be useful in determining the elapsed time of an event. If ticks are requested, the clock tick rate in ticks per second is returned in the most significant word of D3. The least significant word contains the current tick.

The following chart illustrates the values returned in the registers:

	Register Offset	Gregorian Format	Julian Format
d0.l	byte 3	zero	seconds since midnight (long) 0-86399
	2	hour (0-23)	
	1	minute (0-59)	
	0	second (0-59)	
d1.l	byte 2-3	year (integer)	julian day number (long)
	1	month (1-12)	
	0	day (1-31)	

CROSS REFERENCE: See F$STime and F$Julian.

CAVEATS: F$Time will return a date and time of zero (with no error) if no previous call to F$STime has been made. A tick rate of zero indicates the clock is not running.

F$TLink **Install user trap handling module** **F$TLink**

ASSEMBLER CALL: OS9 F$TLink

INPUT: d0.w = User Trap Number (1-15)
d1.l = Optional memory override
(a0) = Module name pointer
If (a0) = 0 or [(a0)] = 0, trap handler is unlinked
Other parameters may be required for specific trap handlers.

OUTPUT: (a0) = Updated past module name
(a1) = Trap library execution entry point
(a2) = Trap module pointer
Other values may be returned by specific trap handlers

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: User traps may be used as a convenient way to link into a standard set of library routines at execution time. This provides the advantage of keeping user programs small, and automatically updating programs that use the library code if it is changed (without having to recompile or relink the program itself). Most Microware utilities use one or more trap libraries.

The F$TLink call attempts to link, or load the named module, installing a pointer to it in the user's process descriptor for subsequent use in trap calls. If a trap module already exists for the specified trap code, an error is returned. OS-9 allocates and initializes static storage for the trap handler if any is required. Traps may be removed by passing a null pointer.

A user program calls a trap routine using the following assembly language directive:

```
tcall N,Function
```

This is the equivalent to:

```
trap #N
dc.w Function
```

"N" can be 1 to 15 (specifying which user trap vector to use). The function code is not used by OS-9, except that it is passed to the trap handler, and the program counter is skipped past it.

F$TLink — Install user trap handling module (continued) — F$TLink

TLink allows the program to delay installation of the handler until a trap is actually used in the program. If a user program executes a user trap call before the corresponding TLink call has been made, the system will execute the user's default trap exception entry point (specified in the module header) if one exists.

CROSS REFERENCE: See Chapter 13 on Trap Handlers.

F$UAcct User Accounting F$UAcct

ASSEMBLER CALL: OS9 F$CCtl

INPUT: d0.w = Function code (F$Fork, F$Chain, F$Exit)
(a0) = Process descriptor pointer
(a3) = User accounting vsect data pointer
(a6) = System global data pointer

OUTPUT: none

ERROR OUTPUT: cc = carry bit set
d1.w = error code if error

FUNCTION: F$UAcct is not implemented in the OS-9 kernel and is not intended to be called by user supplied routines. It is instead, a "hook" that allows system administrators to provide their own user accounting module. The F$UAcct request is made by the kernel whenever a F$Fork, F$Chain or F$Exit system call is processed.

A user accounting module may be provided by creating a System Object type module as part of the OS9Boot file, using the "OS9P2" hook provided by the "init" module. The main execution entry point of this module should install an F$UAcct system call (using the F$SetSvc call), which makes up most of the body of the user accounting module. Note that any static storage allocated for the user accounting module will be passed to the execution entry point in register (a3). This can automatically be passed to the F$UAcct system call at execution time.

F$UAcct is called by fork and chain after the new process' primary module has been located and its process descriptor and data area have been initialized. It is invoked at exit before the system resources have been returned, in particular, before the standard I/O paths have been closed.

It is important to remember that the user accounting module is invoked in system state before and after every process is executed. The F$UAcct request should be kept as fast as possible, since it can significantly effect the overhead of process creation.

If an error (other than E$UnkSvc) is returned by F$UAcct during a fork or chain request, the kernel will abort the new process, thereby denying access to it.

NOTE: This system call must preserve all registers.

F$UnLink **Unlink a module by address** **F$UnLink**

ASSEMBLER CALL: OS9 F$UnLink

INPUT: (a2) = Address of the module header

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$UnLink tells OS-9 that the module is no longer needed by the calling process. The module's link count is decremented. When the link count equals zero, the module is removed from the module directory and its memory deallocated. When several modules are loaded together as a group, modules are only removed when the link count of all modules in the group have zero link counts.

Device driver modules in use and certain system modules can not be unlinked.

CROSS REFERENCE: See F$UnLoad.

CAVEATS: If a bad address is passed, UnLink will NOT find a module in the module directory and will not return an error.

Repetitive UnLink calls to the same module will artificially lower its link count, regardless of how many users are currently using it. If the link count becomes zero while the module is being used, it will be removed from the module directory and its memory deallocated. This will cause severe problems for whoever is currently using the module and may crash the system.

F$UnLoad **Unlink module by name** **F$UnLoad**

ASSEMBLER CALL: OS9 F$UnLoad

INPUT: d0.w = Module type/language
(a0) = Module name pointer

OUTPUT: (a0) = Updated past module name

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$UnLoad locates the module in the module directory, decrements its link count, and removes it from the directory if the count reaches zero. Note that this call differs from F$UnLink in that the pointer to the module name is supplied rather than the address of the module header.

CROSS REFERENCE: See F$UnLink.

CAVEAT: Repetitive UnLoad calls to the same module will artificially lower its link count, regardless of how many users are currently using it. If the link count becomes zero while the module is being used, it will be removed from the module directory and its memory deallocated. This will cause severe problems for whoever is currently using the module and may crash the system.

F$Wait **Wait for child process to terminate** **F$Wait**

ASSEMBLER CALL: OS9 F$Wait

INPUT: None

OUTPUT: d0.w = Terminating child process' ID
d1.w = Child process' exit status code

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$NoChld

FUNCTION: F$Wait causes the calling process to deactivate until a child process terminates by executing an F$Exit system call, or otherwise is terminated. The child's ID number and exit status are returned to the parent. If the child process died due to a signal, the exit status word (d1 register) is the signal code.

If the caller has several children, the caller is activated when the first one dies, so one Wait system call is required to detect termination of each child.

If a child died before the Wait call, the caller is reactivated immediately. Wait returns an error only if the caller has no children.

CROSS REFERENCE: See F$Exit, F$Send and F$Fork.

CAVEATS: The process descriptors for child processes are not returned to free memory until their parent process does an F$Wait system call or terminates.

If a signal is received by a process waiting for children to terminate, it will be activated. In this case, d0.w will contain zero, since no child process has terminated.

NOTES

I$Attach **Attach a new device to the system** **I$Attach**

ASSEMBLER CALL: OS9 I$Attach

INPUT: d0.b = Access mode (Read_, Write_, Updat_)
(a0) = Device name pointer

OUTPUT: (a2) = Address of the device table entry.

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$DevOvf, E$BMode, E$DevBsy, E$MemFul

FUNCTION: I$Attach causes an I/O device to become "known" to the system. It is used to attach a new device to the system, or verify that it is already attached.

The device's name string is used to search the system module directory to see if a device descriptor module with the same name is in memory (this is the name the device is known by). The descriptor module will contain the name of the device's file manager, device driver and other related information.

If the descriptor is found and the device is not already attached, OS-9 will link to its file manager and device driver. It then places their addresses in a new device table entry. Any permanent storage needed by the device driver is allocated, and the driver's initialization routine is called to initialize the hardware.

If the device has already been attached, it will not be reinitialized.

The access mode parameter may be used to verify that subsequent read and/or write operations will be permitted. An Attach system call is not required to perform routine I/O. It does not "reserve" the device in question. It just prepares it for subsequent use by any process.

The kernel attaches all devices at open, and detaches them at close.

NOTE: Attach and Detach for devices are like Link and Unlink for modules; they are usually used together. However, system performance can be improved slightly if all devices are attached at startup. This increments each device's use count and prevents the device from being reinitialized every time it is opened. This also has the advantage of allocating the static storage for devices all at once, which minimizes memory fragmentation. If this is done, the device driver termination routine will never be executed.

CROSS REFERENCE: See I$Detach.

I$ChgDir **Change working directory** **I$ChgDir**

ASSEMBLER CALL: OS9 I$ChgDir

INPUT: d0.b = Access mode (read/write/exec)
(a0) = Address of the pathlist

OUTPUT: (a0) = Updated past pathname

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNam, E$BMode

FUNCTION: ChgDir changes a process' working directory to another directory file specified by the pathlist. Depending on the access mode given, either the execution or the data directory (or both) may be changed. The file specified must be a directory file, and the caller must have access permission for the specified mode.

ACCESS MODES: 1 = Read
2 = Write
3 = Update (read and write)
4 = Execute

If the access mode is read, write, or update the current data directory is changed. If the access mode is execute, the current execution directory is changed. Both may be changed simultaneously.

NOTE: The shell "CHD" directive uses UPDATE mode, which means you must have both read and write permission to change directories from the shell. This is a recommended practice.

I$Close **Close a path to a file/device** **I$Close**

ASSEMBLER CALL: OS9 I$Close

INPUT: d0.w = Path number

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum

FUNCTION: F$Close terminates the I/O path specified by the path. The path number will no longer be valid for any OS-9 calls unless it becomes active again through an Open, Create, or Dup system call. When pathlists to devices that are non-sharable are closed, the devices become available to other requesting processes. If this is the last use of the path (i.e., it has not been inherited or duplicated by I$Dup) all OS-9 internally managed buffers and descriptors are deallocated.

NOTE: The OS9 F$Exit service request automatically closes any open paths. Standard I/O paths are by convention not closed except when it is desired to change the files/devices they correspond to.

CROSS REFERENCE: See I$Detach.

CAVEATS: I$Close does an implied I$Detach call. If this causes the device use count to become zero, the device termination routine will be executed.

I$Create **Create a path to a new file** **I$Create**

ASSEMBLER CALL: OS9 I$Create

INPUT: d0.b = Access mode (S, I, E, W, R)
d1.w = File attributes (access permission)
d2.l = Initial allocation size (optional)
(a0) = Pathname pointer

OUTPUT: d0.w = Path number
(a0) = Updated past the pathlist

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$PthFul, E$BPNam

FUNCTION: F$Create is used to create a new file. On multi-file devices, the new file name is entered in the directory structure. On non-multi-file devices, Create is synonymous with Open.

The access mode parameter passed in register d0.b must have the write bit set if any data is to be written to the file. The file is given the attributes passed in the register d1.w. The individual bits are defined as follows

Mode Bits (d0.w)	Attribute Bits (d1.w)
0 = read	0 = owner read permit
1 = write	1 = owner write permit
2 = execute	2 = owner execute permit
5 = initial file size	3 = public read permit
6 = single user	4 = public write permit
	5 = public execute permit
	6 = non-sharable file

If the execute bit (bit 2) of the access mode byte is set, directory searching will begin with the working execution directory instead of the working data directory.

The path number returned by OS-9 is used to identify the file in subsequent I/O service requests until the file is closed.

File space is allocated for the file automatically by WRITE or explicitly by the SETSTAT call. If the size bit (bit 5) is set, an initial file size estimate may be passed in d2.l.

An error will occur if the pathlist specifies a file name that already exists. I$Create can not be used to make directory files (see I$MakDir).

Create causes an implicit I$Attach call. If the device has not previously been attached the device's initialization routine will be executed.

CROSS REFERENCE: See I$Attach, I$Open, I$Close and I$MakDir.

CAVEATS: The caller is made the owner of the file. Because of compatability with OS-9/6809 disk formats, there is only space for two bytes of owner ID. The LS byte of the user's group and the LS byte of the user's ID are used as the owner ID. All user's with the same group ID may access the file as the owner.

I$Delete **Delete a file** **I$Delete**

ASSEMBLER CALL: OS9 I$Delete

INPUT: d0.b = Access mode (read/write/exec)
(a0) = Pathname pointer

OUTPUT: (a0) = Updated past pathlist

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNam

FUNCTION: This service request deletes the file specified by the pathlist. The caller must have non-sharable write access to the file (the file may not already be open) or an error will result. Attempts to delete non-multifile devices will result in an error.

The access mode is used to specify the data or execution directory (but not both) in the absence of a full pathlist. If the access mode is read, write, or update, the current data directory is assumed. If the execute bit is set, the current execution directory is assumed. Note that if a full pathlist is given (a pathlist beginning with '/'), the access mode is ignored.

CROSS REFERENCE: See I$Detach, I$Attach, I$Create and I$Open.

I$Detach **Remove a device from the system** **I$Detach**

ASSEMBLER CALL: OS9 I$Detach

INPUT: (a2) = Address of the device table entry.

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: I$Detach removes a device from the system device table if not in use by any other process. If this is the last use of the device, the device driver's termination routine is called, and any permanent storage assigned to the driver is deallocated. The device driver and file manager modules associated with the device are unlinked and may be lost if not in use by another process. It is crucial for the termination routine to remove the device from the IRQ system.

The I$Detach service request must be used to un-attach devices that were attached with the I$Attach service request. Both of these are used mainly by the kernel and are of limited use to the typical user. SCF also uses Attach/Detach to setup its second (echo) device.

Most devices are attached at startup and remain attached. Seldom used devices can be attached to the system and used for a while, then detached to free system resources when no longer needed.

CROSS REFERENCE: See I$Attach and I$Close.

CAVEATS: If an invalid address is passed in (a2), the system may crash or undergo severe damage.

I$Dup **Duplicate a path** **I$Dup**

ASSEMBLER CALL: OS9 I$Dup

INPUT: d0.w = Path number of path to duplicate

OUTPUT: d0.w = New number for the same path

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$PthFul, E$BPNum

FUNCTION: Given the number of an existing path, I$Dup returns a synonymous path number for the same file or device. I$Dup will always use the lowest available path number. For example, if the user does I$Close on path #0, then does I$Dup on path #4, then path #0 will be returned as the new path number. In this way, the standard I/O paths may be manipulated to contain any desired paths.

SHELL uses this service request when it redirects I/O. Service requests using either the old or new path numbers operate on the same file or device.

CAVEATS: This only increments the "use count" of a path descriptor and returns a synonymous path number. The path descriptor is NOT copied. It is usually not a good idea for more than one process to be doing I/O on the same path concurrently. On RBF files, unpredictable results may occur.

I$GetStt **Get file/device status** **I$GetStt**

ASSEMBLER CALL: OS9 I$GetStt

INPUT: d0.w = Path number
d1.w = Function code
Others = dependent on function code

OUTPUT: Dependent on function code

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum

FUNCTION: This is a "wild card" call used to handle individual device parameters that are not uniform on all devices or are highly hardware dependent. The exact operation of this call depends on the device driver and file manager associated with the path.

A typical use is to determine a terminal's parameters (echo on/off, delete character, etc.). It is commonly used in conjunction with the SerStt call, which sets the device operating parameters.

The mnemonics for the status codes are found in the relocatable library "sys.l" or "usr.l". Codes 0-127 are reserved for Microware use. The remaining codes and their parameter passing conventions are user definable (see Chapter 7 on device drivers). Below are the presently defined function codes:

MNEMONIC	FUNCTION
SS_Opt	Read PD_OPT: the path descriptor option section. (All)
SS_Ready	Test for data ready. (RBF,SCF, PIPE)
SS_Size	Return current file size. (RBF, PIPE)
SS_Pos	Get current file position. (RBF, PIPE)
SS_EOF	Test for end of file. (RBF,SCF, PIPE)
SS_DevNm	Return device name. (ALL)
SS_FD	Read file descriptor sector. (RBF, PIPE)
SS_FDInf	Get specified file descriptor sector (RBF)

Parameter Passing Conventions

SS_Opt: Read option section of the path descriptor	
INPUT	d0.w = Path number d1.w = #SS_Opt function code (a0) = Address of place to put a 128 byte status packet
OUTPUT	Status packet copied to buffer
ERROR OUTPUT	cc = Carry bit set d1.w = Appropriate error code
FUNCTION	This function reads the option section of the path descriptor and copies it into the 128 byte area pointed to by (a0). It is typically used to determine the current settings for echo, auto line feed, etc. For a complete description of the status packet, see Chapter 7 and 8 on file manager path descriptors.

SS_Ready: Test for data available	
INPUT	d0.w = Path number d1.w = #SS_Ready function code
OUTPUT	d1.l = Number of input characters available on SCF or pipe devices. RBF devices always return carry clear, d1.l=1
ERROR OUTPUT	cc = Carry bit set d1.w = Appropriate error code (E$NRdy if no data available)

SS_Size: Get current file size (RBF or Pipe files)	
INPUT	d0.w = Path number d1.w = #SS_Size function code
OUTPUT	d2.l = Current file size
ERROR OUTPUT	cc = Carry bit set d1.w = Appropriate error code

SS_Pos: Get current file position (RBF or Pipe files)	
INPUT	d0.w = Path number d1.w = #SS_Pos function code
OUTPUT	d2.l = Current file position
ERROR OUTPUT	cc = Carry bit set d1.w = Appropriate error code

SS_EOF: Test for end of file	
INPUT	d0.w = Path number d1.w = #SS_EOF function code
OUTPUT	d1.l = 0 If not EOF, (SCF never returns EOF)
ERROR OUTPUT	cc = Carry bit set d1.w = Appropriate error code (E$EOF, if end of file)

SS_DevNm: Return device name	
INPUT	d0.w = Path number d1.w = #SS_DevNm function code (a0) = Address of 32 byte area for device name
OUTPUT	Device name in 32 byte storage area, null terminated

SS_FD: Read FD sector (RBF or Pipe files)	

INPUT	d0.w = Path number
	d1.w = #SS_FD function code
	d2.w = Number of bytes to copy (<=256)
	(a0) = Address of buffer area for FD

OUTPUT	File descriptor copied into buffer

SS_FDInf: Get specified FD sector (RBF)	

INPUT	d0.w = Path number
	d1.w = #SS_FD function code
	d2.w = Number of bytes to copy (<=256)
	d3.l = FD sector address
	(a0) = Address of buffer area for FD

OUTPUT	File descriptor copied into buffer

I$MakDir **Make a new directory** **I$MakDir**

ASSEMBLER CALL: OS9 I$MakDir

INPUT: d0.b = Access mode (see below)
d1.w = Access permissions
d2.l = Initial Allocation Size (Optional)
(a0) = Pathname pointer

OUTPUT: (a0) = Updated past pathname

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNam, E$CEF

FUNCTION: I$MakDir is the only way a new directory file can be created. It will create and initialize a new directory as specified by the pathlist. The new directory file contains no entries, except for an entry for itself (".") and its parent directory (".."). Makdir will fail on non-multi-file devices. If the execution bit is set, OS-9 will begin searching for the file in the working execution directory (unless the pathlist begins with a slash).

The caller is made the owner of the directory. MakDir does not return a path number because directory files are not "opened" by this request (use I$Open to do so). The new directory will automatically have its "directory" bit set in the access permission attributes. The remaining attributes are specified by the bytes passed in register d1.w which have individual bits defined as below (if the bit is set, access is permitted):

Mode Bits (d0.b)	Attribute Bits (d1.w)
0 = read	0 = owner read permit
1 = write	1 = owner write permit
2 = execute	2 = owner execute permit
5 = initial directory size	3 = public read permit
7 = directory	4 = public write permit
	5 = public execute permit
	6 = non-sharable file
	7 = directory

I$Open **Open a path to a file or device** **I$Open**

ASSEMBLER CALL: OS9 I$Open

INPUT: d0.b = Access mode (D S E W R)
(a0) = Pathname pointer

OUTPUT: d0.w = Path number
(a0) = Updated past pathname

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$PthFul, E$BPNam, E$Bmode, E$FNA, E$PNNF and E$Share.

FUNCTION: I$Open opens a path to an existing file or device as specified by the pathlist. A path number is returned which is used in subsequent service requests to identify the path. If the file does not exist, an error is returned.

The access mode parameter specifies which subsequent read and/or write operations are permitted as follows (if the bit is set, access is permitted):

Mode Bits
0 = read
1 = write
2 = execute
6 = open file for non sharable use
7 = open directory file

NOTE: A file may be opened with no bits set. This allows the user to examine the attributes, size, etc. by the GetStt system call, but does not permit any actual I/O on the path.

For RBF devices, Read mode should be used instead of Update if the file is not going to be modified. This inhibits record locking, and can dramatically improve system performance if more than one user is accessing the file. The access mode must conform to the access permissions associated with the file or device (see I$Create).

If the execution bit mode is set, OS-9 will begin searching for the file in the working execution directory (unless the pathlist begins with a slash).

If the single user bit is set, the file will be opened for non-sharable access even if the file is sharable.

Files can be opened by several processes (users) simultaneously. Devices have an attribute that specifies whether or not they are sharable on an individual basis.

Open will always use the lowest path number available for the process.

CROSS REFERENCE: See I$Attach, I$Create and I$Close.

CAVEATS: Directory files may be opened only if the Directory bit (bit 7) is set in the access mode.

I$Read **Read data from a file or device** **I$Read**

ASSEMBLER CALL: OS9 I$Read

INPUT: d0.w = Path number
d1.l = Maximum number of bytes to read
(a0) = Address of input buffer

OUTPUT: d1.l = Number of bytes actually read

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum, E$Read, E$BMode, E$EOF

FUNCTION: I$Read reads a specified number of bytes from the path number given. The path must previously have been opened in Read or Update mode. The data is returned exactly as read from the file/device without additional processing or editing such as backspace, line delete, etc. If there is not enough data in the file to satisfy the read request, fewer bytes will be read than requested, but an end of file error is not returned.

After all data in a file has been read, the next I$Read service request will return an end of file error.

CROSS REFERENCE: See I$ReadLn

CAVEATS: The keyboard X-ON, X-OFF characters may be filtered out of the input data on SCF-type devices unless the corresponding entries in the path descriptor have been set to zero. It may be desirable to modify the device descriptor so that these values in the path descriptor are initialized to zero when the path is opened. SCF devices usually terminate the read when a carriage return is reached.

For RBF devices, if the file is open for Update, the record read will be locked out. See the Record Locking section in Chapter 8.

The number of bytes requested will be read unless:

A. The end-of-file is reached
B. An end-of-record occurs (SCF only)
C. An error condition occurs

I$ReadLn **Read a text line with editing** **I$ReadLn**

ASSEMBLER CALL: OS9 I$ReadLn

INPUT: d0.w = Path number
d1.l = Maximum number of bytes to read
(a0) = Address of input buffer

OUTPUT: d1.l = Actual number of bytes read

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum, E$Read, E$BMode

FUNCTION: ReadLn is similar to "Read" except it reads data from the input file or device until an end-of-line character is encountered. ReadLn also causes line editing to occur on SCF-type devices. Line editing refers to backspace, line delete, echo, automatic line feed, etc. Some devices (SCF) may limit the number of bytes that may be read with one call.

SCF requires that the last byte entered be an end-of-record character (normally carriage return). If more data is entered than the maximum specified, it will not be accepted and a PD_OVF character (normally bell) will be echoed. For example, a ReadLn of exactly one byte will accept only a carriage return to return without error and beep when other keys are pressed.

After all data in a file has been read, the next I$ReadLn service request will return an end of file error.

CROSS REFERENCE: See I$Read.

I$Seek — Reposition the logical file pointer — I$Seek

ASSEMBLER CALL: OS9 I$Seek

INPUT: d0.w = Path number
d1.l = New position

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum

FUNCTION: I$Seek repositions the path's "file pointer"; which is the 32-bit address of the next byte in the file to be read or written. I$Seek usually does not initiate any physical positioning of the media.

A Seek may be performed to any value even if the file is not large enough. Subsequent Writes will automatically expand the file to the required size (if possible), but Reads will return an end-of-file condition. Note that a Seek to address zero is the same as a "rewind" operation.

Seeks to non-random access devices are usually ignored and return without error.

CAVEATS: On RBF devices, seeking to a new disk sector causes the internal sector buffer to be rewritten to disk if it has been modified. Seek does not change the state of record locks. Beware of seeking to a negative position. RBF will take negatives as large positive numbers.

I$SetStt — Set file/device status — I$SetStt

ASSEMBLER CALL: OS9 I$SetStt

INPUT: d0.w = Path number
d1.w = Function code
Others = Function code dependent

OUTPUT: Function code dependent

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum

FUNCTION: This is a "wild card" system call used to handle individual device parameters that are not uniform on all devices or are highly hardware dependent. The exact operation of this call depends on the device driver and file manager associated with the path.

A typical use is to set a terminal's parameters for backspace character, delete character, echo on/off, null padding, paging, etc. It is commonly used in conjunction with the GetStt service request which is used to read the device's operating parameters. Below are the presently defined function codes:

MNEMONIC	FUNCTION
SS_Opt	Write the options section of the path descriptor (ALL)
SS_Size	Set the file size (RBF, PIPE)
SS_Reset*	Restore head to track zero (SBF)
SS_WTrk*	Write (format) track
SS_Frz	Freeze DD_ information (currently not implemented)
SS_SPT	Set Sectors per track (currently not implemented)
SS_SQD*	Sequence down disk drive (SBF)
SS_DCmd	Direct command to hard disk controller (currently not implemented)

* These setstats exist in Microware disk drivers (if needed).

MNEMONIC	FUNCTION
SS_FD	Write FD sector (RBF)
SS_Ticks	Set Record lockout honor duration (RBF)
SS_Lock	Lock/Release record (RBF)
SS_SSig	Send signal on data ready (SCF,PIPE)
SS_Relea	Release device (SCF,PIPE)
SS_Attr	Set file attributes (RBF,PIPE)
SS_EnRTS	Enable RTS line
SS_DsRTS	Disable RTS line
SS_DCOn	Sends signal when Data Carrier Detect line goes true
SS_DCOff	Sends signal when Data Carrier Detect line goes false
SS_Feed	Erase tape (SBF)
SS_WFM	Write tape mark(s) (SBF)
SS_RFM	Skip past tape mark(s) (SBF)
SS_Skip	Skips block(s) (SBF)

NOTE: Only SS_Reset and SS_WTrk are required. Codes 128 through 255 and their parameter passing conventions are user definable (see the sections of this manual on writing device drivers). The function code and register stack are passed to the device driver.

Parameter Passing Conventions

SS_Opt: Write option section of path descriptor	
INPUT	d0.w = Path number d1.w = #SS_Opt function code (a0) = Address of a 128 byte status packet
OUTPUT	none
FUNCTION	This writes the option section of the path descriptor from the 128 byte status packet pointed to by (a0). It is typically used to set the device operating parameters (echo, auto line feed, etc.). This call is handled by the file managers, and only copies values that are appropriate to be changed by user programs.

SS_Size: Set file size (RBF,PIPE)	

INPUT	d0.w = Path number
	d1.w = #SS_Size function code
	d2.l = Desired file size

OUTPUT	none

SS_Reset: Restore head to track zero (RBF, SBF)	
INPUT	d0.w = Path number d1.w = #SS_Reset function code
OUTPUT	none
FUNCTION	For RBF, this directs the disk head to track zero. It is used for formatting and for error recovery. For SBF, this rewinds the tape.

SS_WTrk: Write track (RBF)	
INPUT	d0.w = Path number d1.w = #SS_WTrk function code (a0) = Address of track buffer (a1) = Address of interleave table This table contains byte entries of LSNs ordered to match the requested interleave offset. d2 = Track number d3.w = Side/density The low order byte of (d3) has 3 settable bits: Bit 0 = SIDE (0 = side zero, 1 = side one) Bit 1 = DENSITY (0 = single, 1 = double) Bit 2 = TRACK DENSITY (0 = single, 1 = double) The high order byte contains the side number. d4 = Interleave value
OUTPUT	none
FUNCTION	This causes a format track operation (used with most floppy disks) to occur. For hard or floppy disks with a "format entire disk" command, this will format the entire media only when the track number equals zero and the side byte equals zero.

SS_FD: Write FD Sector (RBF)	
INPUT	d0.w = Path Number d1.w = #SS_FD function code (a0) = Address of FD sector image
OUTPUT	none
FUNCTION	This changes FD sector data. The path must be open for write. NOTE: Only FD_OWN, FD_DAT, and FD_Creat can be changed. These are the only fields written back to disk. Only the super user can change the file's owner ID.

SS_Ticks: Wait specified number of ticks for record release.(RBF)	

INPUT	d0.w = path number
	d1.w = #SS_Ticks function code
	d2.l = Delay interval

OUTPUT	none
FUNCTION	Normally, if a read or write request is issued for a part of a file that is locked out by another user, RBF sleeps indefinitely until the conflict is removed. The SS_Ticks call may be used to cause an error (E$Lock) to be returned to the user program if the conflict still exists after the specified number of ticks have elapsed. The delay interval is used directly as a parameter to RBF's conflict sleep request. The value zero (RBF's default) causes a sleep forever until the record is released. A value of one means that if the record is not released immediately, an error is returned. If the high order bit is set, the lower 31 bits are converted from 256th of a second into ticks before sleeping. This allows programmed delays to be independent of the system clock rate.

SS_Lock: Lock out a record (RBF)	

INPUT	d0.w = Path Number
	d1.w = #SS_Lock function code
	d2.l = Lockout size

OUTPUT	none
FUNCTION	SS_Lock locks out a section of the file from the current file pointer position up to the specified number of bytes. If 0 bytes are requested, all locks are removed (Record Lock, EOF Lock, and File Lock). If $FFFFFFFF bytes are requested, then the entire file is locked out regardless of where the file pointer is. This is a special type of "file lock" that remains in effect until released by SS_Lock(0), a read or write of zero bytes, or the file is closed. There is no way to gain file lock using only Read or Write system calls.

SS_SSig: Send Signal on data ready (SCF,PIPE)	
INPUT	d0.w = Path number d1.w = SS_SSig function code d2.w = User defined signal code
OUTPUT	none
FUNCTION	SS_SSig sets up a signal to be sent to a process when an interactive device or pipe has data ready. SS_SSig must be reset each time the signal is sent. The device or pipe is considered busy and returns an error if any read request arrives before the signal is sent. Write requests to the device are allowed in this state.
CAVEATS	If an unprocessed signal is pending when a character is received, the SS_SSig signal will be lost. Programs that use this call must be wary of this situation.

SS_Relea: Release device (SCF,PIPE)	
INPUT	d0.w = path number d1.w = SS_Relea function code
OUTPUT	none
FUNCTION	SS_Relea releases the device from any SS_SSig, SS_DCOn or SS_DCOff request made by the calling process.

SS_Attr: Set the file attributes (RBF,PIPE)	

INPUT	d0.w = Path number
	d1.w = #SS_Attr function code
	d2.w = New attributes

OUTPUT	none
FUNCTION	This changes a file's attributes to the new value if possible. It is not permitted to set the dir bit of a non-directory file, or to clear the dir bit of a non empty directory.

SS_EnRTS: Enables RTS line	
INPUT	d0.w = path number d1.w = SS_EnRTS function code
OUTPUT	none

SS_DsRTS: Disables RTS line	
INPUT	d0.w = path number d1.w = SS_DsRTS function code
OUTPUT	none

SS_DCOn: Sends signal when Data Carrier Detect line goes true	
INPUT	d0.w = path number d1.w = SS_DCOn function code d2.w = Signal code to be sent
OUTPUT	none
FUNCTION	When a modem receives a carrier, the Data Carrier Detect line will go true. SS_DCOn will send a signal code when this happens. SS_DCOff will send a signal when the line goes false.

SS_DCOff: Sends signal when Data Carrier Detect line goes false	
INPUT	d0.w = path number d1.w = SS_DCOff function code d2.w = Signal code to be sent
OUTPUT	none
FUNCTION	When a modem has finished receiving data from a carrier, the Data Carrier Detect line will go false. SS_DCOff will send a signal code when this happens. SS_DCOn will send a signal when the line goes true.

SS_Feed: Erase tape (SBF)	
INPUT	d0.w = path number d1.w = SS_Feed function code d2.l = # of blocks to erase
OUTPUT	none
FUNCTION	This will erase a portion of the tape. The amount of tape erased will depend on the capabilities of the hardware used. SBF will attempt to use the following: If -1 is passed in d2, SBF will erase until the end-of-tape is reached. If d2 recieves a positive parameter, SBF will erase the amount of tape equivalent to that number of blocks. Again this will be dependent on both the hardware used and the driver.

SS_WFM: Write tape marks (SBF)	
INPUT	d0.w = path number d1.w = SS_WFM function code d2.l = # of tape marks
OUTPUT	none
FUNCTION	This will write the number of tape marks specified in d2.

SS_RFM: Skip tape marks (SBF)	
INPUT	d0.w = path number d1.w = SS_RFM function code d2.l = # of tape marks
OUTPUT	none
FUNCTION	This will skip the number of tape marks specified in d2. If d2 is negative, the tape will be rewound the specified number of marks.

SS_Skip: Skip blocks (SBF)	
INPUT	d0.w = path number d1.w = SS_Relea function code d2.l = # of blocks to skip
OUTPUT	none
FUNCTION	This will skip the number of blocks specified in d2. If the number is negative, the tape will be rewound the specified number of blocks.

I$Write **Write data to a file or device** **I$Write**

ASSEMBLER CALL: OS9 I$Write

INPUT: d0.w = Path number
d1.l = Number of bytes to write
(a0) = Address of buffer

OUTPUT: d1.l = Number of bytes actually written

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum, E$BMode, E$Write

FUNCTION: I$Write outputs bytes to a file or device associated with the path number specified. The path must have been OPENed or CREATEd in the Write or Update access modes.

Data is written to the file or device without processing or editing. If data is written past the present end-of-file, the file is automatically expanded.

CROSS REFERENCE: See I$Open, I$Create and I$WritLn

CAVEATS: On RBF devices, any record that was locked is released.

I$WritLn **Write a line of text with editing** **I$WritLn**

ASSEMBLER CALL: OS9 I$WritLn

INPUT: d0.w = Path number
d1.l = Maximum number of bytes to write
(a0) = Address of buffer

OUTPUT: d1.l = Actual number of bytes written

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$BPNum, E$BMode, E$Write

FUNCTION: This system call is similar to Write except it writes data until a carriage return character or (d1) bytes are encountered. Line editing is also activated for character-oriented devices such as terminals, printers, etc. The line editing refers to auto line feed, null padding at end-of-line, etc.

The number of bytes actually written (returned in d1.l) does not reflect any additional bytes that may have been added by file managers or device drivers for device control. For example, if SCF appends a line feed and nulls after carriage return characters, these extra bytes will not be counted.

CROSS REFERENCE: See I$Open, I$Create, I$Write and Chapter 7 on SCF Drivers (line editing).

CAVEATS: On RBF devices, any record that was locked is released.

F$AllPD **Allocate process/path descriptor** **F$AllPD**

ASSEMBLER CALL: OS9 F$AllPD

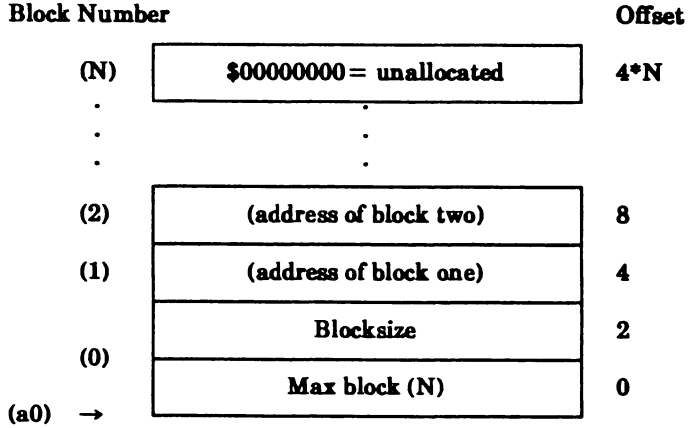
INPUT: (a0)=process/path table pointer

OUTPUT: d0.w=process/path number
(a1)=pointer to process/path descriptor

ERROR OUTPUT: cc = Carry bit set
d1.w = error code if error

FUNCTION: F$AllPd is used to dynamically allocate fixed length blocks of system memory. It allocates and initializes (to zeros) a block of storage and return its address.

It can be used with F$FindPD and F$RetPD to perform simple memory management. The system uses these routines to keep track of memory blocks used for process and path descriptors. They can be used generally for similar purposes by creating a map table for the data allocations. The table must be initialized as follows:



CROSS REFERENCE: See F$FindPD and F$RetPD.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$AllPrc **Allocate process descriptor** **F$AllPrc**

ASSEMBLER CALL: OS9 F$AllPrc

INPUT: None

OUTPUT: (a2) = Process Descriptor pointer

ERROR OUTPUT: cc = Carry bit set.
d1.w = Appropriate error code.

POSSIBLE ERRORS: E$PrcFul

FUNCTION: F$AllPrc allocates and initializes a process descriptor. The address of the descriptor is kept in the process descriptor table. Initialization consists of clearing the descriptor, setting up the state as system state, and marking as unallocated as much of the MMU image as the system allows.

On systems without memory management/protection, this is a direct call to F$AllPD.

CROSS REFERENCE: See F$AllPD

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST.

F$AProc **Insert process in active process queue** **F$AProc**

ASSEMBLER CALL: OS9 F$AProc

INPUT: (a0) = Address of process descriptor

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$AProc inserts a process into the active process queue so that it may be wscheduled for execution. All processes already in the active process queue are aged. The age of the specified process is set to its priority. The process is then inserted according to its relative age. If the new process has a higher priority than the currently active process, the active process will give up the remainder of its time slice and the new process will run immediately.

CAVEATS: OS-9 does not preempt a process that is in system state (i.e. the middle of a system call). However, OS-9 does set a bit in the process descriptor that will cause it to give up its time slice when it reenters user state.

CROSS REFERENCE: See F$NProc and Chapter 4 on Process Scheduling.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$FindPD **Find process/path descriptor** **F$FindPD**

ASSEMBLER CALL: OS9 F$FindPD

INPUT: d0.w=process/path number
(a0)=process/path table pointer

OUTPUT: (a1)=pointer to process/path descriptor

ERROR OUTPUT: cc = Carry bit set
d1.w = error code if error

FUNCTION: F$FindPD converts a process or path number to the absolute address of its descriptor data structure. It can be used for simple memory management of fixed length blocks. See F$AllPD for a description of the data structure used.

CROSS REFERENCE: See F$AllPd and F$RetPd.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$IOQu **Enter I/O queue** **F$IOQu**

ASSEMBLER CALL: OS9 F$IOQu

INPUT: d0.w = Process Number.

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set.
d1.w = Appropriate error code.

FUNCTION: F$IOQu links the calling process into the I/O queue of the specified process and performs an untimed sleep. It is assumed that routines associated with the specified process will send a wakeup signal to the calling process. IOQu is used primarily and extensively by the I/O system.

For example, if a process needs to do I/O on a particular device that is busy servicing another request, the calling process will perform an F$IOQu call to the process in control of the device. When the first process returns from the file manager, the kernel will automatically wake up the IOQu-ed process.

CROSS REFERENCE: See F$FindPd, F$Send and F$Sleep.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$IRQ **Add or remove device from IRQ table** **F$IRQ**

ASSEMBLER CALL: OS9 F$IRQ

INPUT:
```
d0.b = vector number
         25-31 for autovectors
         64-255 for vectored IRQs
d1.b = priority (0 = polled first, 255 = last)
(a0) = IRQ service routine entry point (0 = delete)
(a2) = global static storage pointer (must be unique to device)
(a3) = port address
```

OUTPUT: None

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

POSSIBLE ERRORS: E$POLL is returned if the polling table is full.

FUNCTION: F$IRQ installs an IRQ service routine into the system polling table. If (a0)=0, the call deletes the IRQ service routine, and only (d0/a0/a2) are used.

The port is sorted by priority onto a list of devices for the specified vector. If the priority is zero, only this device will be allowed to use the vector. Otherwise, any vector may support multiple devices. OS-9 does not poll the I/O port prior to calling the interrupt service routine and makes no use of (a3). Device drivers are required to determine if their device caused the interrupt. Service routines conform to the following register conventions:

```
INPUT:        (a2) = global static pointer
              (a3) = port address
              (a6) = system global data pointer (D_'s)
              (a7) = system stack (in active proc's descriptor)

OUTPUT:       None

ERROR OUTPUT: Carry bit set if the device did not cause the interrupt.

WARNING:      Interrupt service routines may destroy the following registers:
              d0, d1, a0, a2, a3 and/or a6.  All other registers used must be
              preserved.
```

CROSS REFERENCE: See Chapter 7 and Chapter 8 for more information on RBF and SCF device drivers.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$Move **Move data (low bound first)** **F$Move**

ASSEMBLER CALL: OS9 F$Move

INPUT: d2.1 = Byte count to copy
(a0) = Source pointer
(a2) = Destination pointer

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$Move is a fast "block-move" subroutine capable of copying data bytes from one address space to another (usually from system to user or vice versa).

The data movement subroutine is optimized to make use of long moves whenever possible. If the source and destination buffers overlap, an appropriate move (left to right or right to left) is used to avoid loss of data due to incorrect propagation.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$NProc **Start next process** **F$NProc**

ASSEMBLER CALL: OS9 F$NProc

INPUT: None

OUTPUT: Control does not return to caller.

ERROR OUTPUT:
```
cc = Carry bit set
d1.w = Appropriate error code
```

FUNCTION: This system call takes the next process out of the Active Process Queue and initiates its execution. If there is no process in the queue, OS-9 waits for an interrupt, and then checks the active process queue again.

CAVEATS: The process calling NProc should already be in one of the system's process queues. If it is not, then it will become unknown by the system even though the process descriptor still exists and will be printed out by a Procs command.

CROSS REFERENCE: See F$AProc.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$DelPrc Deallocate process descriptor service request F$DelPrc

ASSEMBLER CALL: OS9 F$DelPrc

INPUT: d0.w = process ID to deallocate

OUTPUT: none

ERROR OUTPUT: cc = carry set
d1.w = appropriate error code

POSSIBLE ERRORS: E$BNam, E$KwnMod

FUNCTION: F$DelPrc deallocates a process descriptor previously allocated by F$AllPD. It is the caller's responsibility to insure that any system resources used by the process are returned prior to calling F$DelPrc.

Currently, the F$DelPrc request is simply a convenient interface to the F$RetPD service request. It is preferred to F$RetPD to insure compatability with future releases of the operating system, that may need to perform process specific deallocations.

CROSS REFERENCE: See F$AllPrc, F$AllPD, F$FidnPD and F$RetPD

NOTE: THIS IS A SYSTEM STATE SERVICE REQUEST

F$RetPD **Return process/path descriptor** **F$RetPD**

ASSEMBLER CALL: OS9 F$RetPD

INPUT: d0.w = process/path number
(a0) = process/path table pointer

OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$RetPD deallocates a process or path descriptor. It can be used in conjunction with F$AllPD and F$FindPD to perform simple memory management of other fixed length objects.

CROSS REFERENCE: See F$AllPD and F$FindPD.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$SSvc **Service request table initialization** **F$SSvc**

ASSEMBLER CALL: OS9 F$SSvc

INPUT: (a1) = pointer to service request initialization table
(a3) = user defined
OUTPUT: None

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

FUNCTION: F$SSvc is used to add or replace function requests in OS-9's user and privileged system service request tables.

(a3) is intended to point globel static storage this allows a globel data pointer to be associated with each installed system call. Whenever the system call is invoked, the data pointer is automatically passed. (a3) may point to anu thing, however. Whatever (a3) point to, is then passed to system call.

An example initialization table might look like this:

```
SvcTbl
       dc.w F$Service          OS-9 service request code
       dc.w Routine-*-2        offset of routine to process request
            :
       dc.w F$Service+SysTrap  redefine system level request
       dc.w SysRoutin-*-2      offset of routine to handle system request
            :
       dc.w -1 end of table
```

Valid service request codes range from (0-255).

If the sign bit of the function code word is set, only the system table will be updated. Otherwise, both the system and user tables will be updated.

Privileged system service requests may only be called from routines executing in System (supervisor) state. The example above shows how a service call is installed that must behave differently in system state than it does in user state.

System service routines are executed in supervisor state, and are not subject to time sliced task switching. They are written to conform to register conventions shown in the following table:

F$SSvc — Service request table initialization (Continued) — F$SSvc

INPUT	d0-d6 = user's values a0-a2 = user's values (a4) = current process descriptor pointer (a5) = user's registers image pointer (a6) = system global data pointer
OUTPUT	cc = carry set d1.w = error code if error

The service request routine should process its request and return from subroutine with an RTS instruction. Any of the registers d0-d7 and a0-a6 may be destroyed by the routine, although for convenience, a4-a6 are generally left intact.

The user's register stack frame pointed to by (a5) is defined in the library sys.l and follows the natural hardware stacking order. If the cc Carry bit is returned set, the service dispatcher will set R$cc and R$d1.w in the user's register stack. Any other values to be returned to the user must be changed in their stack by the service routine.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

F$VModul **Verify module** **F$VModul**

ASSEMBLER CALL: OS9 F$VModul

INPUT: d0.l = Module group ID
(a0) = Address of module

OUTPUT: (a2) = Directory entry pointer

ERROR OUTPUT: cc = Carry bit set
d1.w = Appropriate error code

POSSIBLE ERRORS: E$KwnMod, E$DirFul, E$BMID, E$BMCRC, E$BMHP

FUNCTION: F$VModul checks the module header parity and CRC bytes of an OS-9 module.
If the header values are valid, the module is entered into the module directory, and a pointer to the directory entry is returned.

The module directory is first searched for another module with the same name. If a module
with the same name and type exists, the one with the highest revision level is retained in the module directory. Ties are broken in favor of the established module.

CROSS REFERENCE: See F$CRC and F$Load.

NOTE: THIS IS A PRIVILEGED SYSTEM STATE SERVICE REQUEST

SYSTEM CALL INDEX

Chapter 16 - User State System Calls

F$AllBit	Allocate in bit map	16-1
F$Chain	Chain process to new module	16-2
F$CmpNam	Compare two names	16-4
F$CpyMem	Copy external memory	16-5
F$CRC	Generate CRC	16-6
F$DatMod	Create a data module	16-7
F$DelBit	Deallocate in bit map	16-8
F$DExec	Execute debugged program	16-9
F$DExit	Exit debugged program	16-11
F$DFork	Fork process under control of debugger	16-12
F$Exit	Terminate process	16-13
F$Fork	Start new process	16-15
F$GModDr	Get module directory copy	16-17
F$GPrDBT	Get process descriptor block table copy	16-18
F$GPrDsc	Get process descriptor copy	16-19
F$ID	Return process ID	16-20
F$Icpt	Set signal intercept	16-21
F$Julian	Get julian date	16-22
F$Link	Link to module	16-23
F$Load	Load module(s) from file	16-24
F$Mem	Set memory size	16-25
F$PErr	Print error message	16-26
F$PrsNam	Parse a path name	16-27
F$RTE	Return from interrupt exception	16-28
F$SchBit	Search bit map	16-29
F$Send	Send signal to process	16-30
F$SetCRC	Generate valid CRC in module	16-31
F$SetSys	Set/examine system global variables	16-32
F$Sleep	Put calling process to sleep	16-33
F$SPrior	Set process priority	16-34
F$SRqMem	System memory request	16-35
F$SRtMem	System memory return	16-36
F$SSpd	Suspend process	16-37
F$STime	Set current time	16-38
F$STrap	Set error Trap handler	16-39
F$SUser	Set user ID number	16-41
F$SysDbg	Call system debugger	16-42
F$Time	Set current date and time	16-43
F$TLink	Install user Trap handling module	16-45
F$UnLink	Unlink module	16-47
F$UnLoad	Unlink module by name	16-48
F$Wait	Wait for child process to terminate	16-49

Chapter 17 - I/O System Calls

I$Attach	Attach I/O device	17-1
I$ChgDir	Change working directory	17-2
I$Close	Close path	17-3
I$Create	Create new file	17-4
I$Delete	Delete file	17-6
I$Detach	Detach I/O device	17-7
I$Dup	Duplicate path	17-8
I$GetStt	Get file/device status	17-9
I$MakDir	Make directory file	17-13
I$Open	Open a path to a file or device	17-14
I$Read	Read data from a file or device	17-16
I$ReadLn	Read line of ASCII data	17-17
I$Seek	Change current position	17-18
I$SetStt	Set file/device status	17-19
I$Write	Write data to file or device	17-29
I$WritLn	Write line of ASCII data	17-30

Chapter 18 - System State System Calls

F$AllPD	Allocate process/path descriptor	18-1
F$AllPrc	Allocate process descriptor	18-2
F$AProc	Enter active process queue	18-3
F$FindPD	Find process/path descriptor	18-4
F$IOQu	Enter I/O queue	18-5
F$IRQ	Add or remove device from IRQ table	18-6
F$Move	Move data (low bound first)	18-7
F$NProc	Start next process	18-8
F$RetPD	Return process/path descriptor	18-9
F$SSvc	Service request table initialization	18-10
F$VModul	Validate module	18-12

End Of Appendix A

EXAMPLES

The examples in this section are to be used as guides in creating your own modules. The examples should be used as samples and should not be assumed to be the most current software. Software for your individual system may be different.

INIT Module Example:

```
Microware OS-9/68000 Resident Macro Assembler V1.8  87/06/15  14:40  Page 1 init.a
Init: OS-9 Configuration Module -
00001                          nam       Init:     OS-9 Configuration Module
00022  0000000b Edition         equ       11        current edition number
00023
00024  00000c00 Typ_Lang       set       (Systm<<8)+0
00025  00008000 Attr_Rev       set       (ReEnt<<8)+0
00026                          psect     init,Typ_Lang,Attr_Rev,Edition,0,0
00027
00028 * Configuration constants (default; changable in "systype.d" file)
00029  000109a0 CPUTyp         set       68000     cpu type (68008/68000/68010)
00030  00000001 Level          set       1         OS-9 Level One
00031  00000002 Vers           set       2         Version 2.1
00032  00000001 Revis          set       1
00033  00000000 Edit           set       0         Edition
00034  00000000 Site           set       0         installation site code
00035  00000080 MDirSz         set       128       initial module directory size (unused)
00036  00000020 PollSz         set       32        IRQ polling table size (fixed)
00037  00000020 DevCnt         set       32        device table size (fixed)
00038  00000040 Procs          set       64        init process tbl size (divisibl by 64)
00039  00000040 Paths          set       64        init path tbl size (divisible by 64)
00040  00000002 Slice          set       2         ticks per time slice
00041  00000080 SysPri         set       128       initial system priority
00042  00000000 MinPty         set       0         initial system min executable priority
00043  00000000 MaxAge         set       0         initial system max natural age limit
00044  00000000 MaxMem         set       0         top of RAM (unused)
00045  00000000 Events          set       0         init event table size (divisible by 8)
00046  00000000 Compat          set       0         version smoothing byte
00047
00048 * Compat flag bit definitions
00049  00000001 SlowIRQ        equ       1         save all regs during IRQ process
00050  00000002 NoStop         equ       2         don't use 'stop' instruction
00051  00000004 NoGhost        equ       4         don't retain Ghost memory moduless
00052
00053                          use       defsfile  (defsfile overrides above definitions)
00001                          opt       f         issue form feeds
00002                          opt       -d96
00003                          opt       -n        no line numbering
00004                          use       /h0/defs/oskdefs.d
00001                          opt       -l
00005                          use       systype.d
00001 * System Definitions for Processor 32 System
00002                          opt       -l
00211
00006
00007
```

INIT Module Example: (continued)

```
Microware OS-9/68000 Resident Macro Assembler V1.8  87/06/15  14:40  Page 2 init.a
Init: OS-9 Configuration Module -
00055 * Configuration module body
00056 0000 0000              dc.l     MaxMem     (unused)
00057 0004 0020              dc.w     PollSz     IRQ polling table size
00058 0006 0020              dc.w     DevCnt     device table size
00059 0008 0040              dc.w     Procs      initial process table size
00060 000a 0040              dc.w     Paths      initial path table size
00061 000c 0096              dc.w     SysParam   param string for first executable mod
00062 000e 008b              dc.w     SysStart   first executable module name offset
00063 0010 007a              dc.w     SysDev     system default device name offset
00064 0012 007e              dc.w     ConsolNm   standard I/O pathlist name offset
00065 0014 00aa              dc.w     Extens     Customization module name offset
00066 0016 0084              dc.w     ClockNm    clock module name offset
00067 0018 0002              dc.w     Slice      number of ticks per time slice
00068 001a 0000              dc.w     0          reserved (formerly UsrAct name offset)
00069 001c 0000              dc.l     Site       installation site code
00070 0020 0064              dc.w     MainFram   installation name offset
00071 0022 0001              dc.l     CPUTyp     specific 68000 family processor in use
00072 0026 0102              dc.b     Level,Vers,Revis,Edit OS-9 Level
00073 002a 0054              dc.w     OS9Rev     OS-9 revision string offset
00074 002c 0080              dc.w     SysPri     initial system priority
00075 002e 0000              dc.w     MinPty     initial system min executable priority
00076 0030 0000              dc.w     MaxAge     maximum system natural age limit
00077 0032 0000              dc.l     MDirSz     module directory size (unused)
00078 0036 0000              dc.w     Events     init event table size (num of entries)
00079 0038   00              dc.b     Compat     version change smooth byte
00080 0039   00              dc.b     0
00081 003a 0000              dc.w     0,0,0,0,0,0    reserved
00082 0046 0000              dc.w     0,0,0,0,0,0,0  reserved
00083
00084 * Configuration name strings
00085 0054 4f53 OS9Rev        dc.b     "OS-9/"
00089 0059 3638              dc.b     "68000"
00091 005e 2056              dc.b     " V",Vers+'0',".",Revis+'0',0
00092
00093 * The remaining names are defined in the "systype.d" macro
00094                         CONFIG
00095 0064 4d69+MainFram     dc.b     "Microware Systems P32",0
00096 007a 2f68+SysDev       dc.b     "/h0",0        initial system disk pathlist
00097 007e 2f74+ConsolNm     dc.b     "/term",0      console terminal pathlist
00098 0084 7033+ClockNm      dc.b     "p32clk",0     clock module name
00099 008b 636d+SysStart     dc.b     "cmds/shell",0 name of initial module to execute
00100 0096=6368+SysParam     dc.b     "chx cmds; startup&",C$CR,0 param to SysStart
00101 00aa 7370+Extens       dc.b     "spu UAcct OS9P2",0 extension module(s)
00102
00106  000000ba              ends
Errors: 00000
Memory used: 13k
Elapsed time: 10 second(s)
```

SCF Descriptor Example

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/04  09:38  Page 1  term.a
Term - 68000 Term device descriptor module
00001                           nam     Term
00002                           ttl     68000           Term device descriptor module
00003                           use     defsfile
00004
00005
00006
00007
00008                           use     ../io/scfdesc.a
00011
00012  00000004 Edition         equ     4               current edition number
00013
00014  00000f00 TypeLang        set     (Devic<<8)+0
00015  00008000 Attr_Rev        set     (ReEnt<<8)+0
00016                           psect   ScfDesc,TypeLang,Attr_Rev,Edition,0,0
00017
00018 0000 00fe                 dc.l    Port            port address
00019 0004   70                 dc.b    Vector          auto-vector trap assignment
00020 0005   02                 dc.b    IRQLevel        IRQ hardware interrupt level
00021 0006   53                 dc.b    Priority        irq polling priority
00022 0007   23                 dc.b    Mode            Device mode capabilities
00023 0008 0034                 dc.w    FileMgr         file manager name offset
00024 000a 0038                 dc.w    DevDrv          device driver name offset
00025 000c=0000                 dc.w    DevCon          (reserved)
00026 000e 0000                 dc.w    0,0,0,0         reserved
00027 0016 001c                 dc.w    OptSiz          option byte count
00028
00029 * Default Parameters
00030           Options
00031 *                                                                 default
00032 *                                 name            function        value
00033 *                                --------        --------------------    -------
00034 0018   00             dc.b     DT_SCF          device type             SCF
00035 0019   00             dc.b     upclock         upcase lock             OFF
00036 001a   01             dc.b     bsb             backspace=BS,SP,BS      ON
00037 001b   00             dc.b     linedel         line del/bsp line       OFF
00038 001c   01             dc.b     autoecho        full duplex             ON
00039 001d   01             dc.b     autolf          auto line feed          ON
00040 001e   00             dc.b     eolnulls        null count              0
00041 001f   00             dc.b     pagpause        end of page pause       OFF
00042 0020   18             dc.b     pagsize         lines per page          24
00043 0021   08             dc.b     C$Bsp           backspace char          ^H
00044 0022   18             dc.b     C$Del           delete line char        ^X
00045 0023   0D             dc.b     C$CR            end of record char      <return>
00046 0024   1B             dc.b     C$EOF           end of file char        ESC
```

SCF Descriptor Example: (continued)

```
00047 0025    04              dc.b    C$Rprt          reprint line char       ^D
00048 0026    01              dc.b    C$Rpet          dup last line char      ^A
00049 0027    17              dc.b    C$Paus          pause char              ^W
00050 0028    03              dc.b    C$Intr          Keyboard Interrupt char ^C
00051 0029    05              dc.b    C$Quit          Keyboard Quit char      ^E
00052 002a    08              dc.b    C$Bsp           backspace echo char     ^H
00053 002b    07              dc.b    C$Bell          line overflow char      ^G
00054 002c    00              dc.b    Parity          stop bits and parity    none
00055 002d    0E              dc.b    BaudRate        bits/char and baud rate none
00056 002e=0000               dc.w    EchoNam         offset of echo device   none
00057 0030    11              dc.b    C$XOn           Transmit Enable char    ^Q
00058 0031    13              dc.b    C$XOff          Transmit Disable char   ^S
00059 0032    09              dc.b    C$Tab           tab character
00060 0033    00              dc.b    tabsize         tab column size
00061  0000001c OptSiz        equ     *-Options
00062
00063 0034 5363 FileMgr       dc.b    "Scf",0         file manager
00080
00081  00000023 Mode          set     ISize_+Updat_   default device mode capabilities
00082
00010  00000040              ends
00011
Errors: 00000
Memory used: 31k
Elapsed time: 26 second(s)
```

EXAMPLES

SBF Descriptor Example:

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/06  14:53  Page     1
 mt0_tm3000.a
MT0 Device Descriptor - Device Descriptor for Tape controller
                    nam     MT0             Device Descriptor

                    use     defsfile
* generic defsfile, no hardware dependent conditions allowed
                    .
                    opt     f               issue form feeds
                    opt     n               no line numbering
                    use     ../defs/oskdefs.d
                    opt     -l

                    use     ../defs/sbfdesc.d

                    ttl     Device          Descriptor for Tape controller


 00000002 Edition   equ     2               current edition number


 00000f00 TypeLang  set     (Devic<<8)+0
 00008000 Attr_Rev  set     (ReEnt<<8)+0
                    psect   SBFDesc,TypeLang,Attr_Rev,Edition,0,0

0000 00ff           dc.l    Port            port address
0004   1d           dc.b    Vector          auto-vector trap assignment
0005   05           dc.b    IRQLevel        IRQ hardware interrupt level
0006   80           dc.b    Priority        irq polling priority
0007   67           dc.b    Mode            device mode capabilities
0008 0026           dc.w    FileMgr         file manager name offset
000a 002a           dc.w    DevDrv          device driver name offset
000c 0000           dc.w    DevCon          (reserved)
000e 0000           dc.w    0,0,0,0         reserved
0016 000e           dc.w    OptLen

* Default Parameters
          OptTbl
0018   03           dc.b    3               DT_SBF device type
0019   00           dc.b    DrvNum          drive number
001a   00           dc.b    0
001b   04           dc.b    NumBlks         maximum number of block buffers
001c 0000           dc.l    BlkSize         block size
0020 03e8           dc.w    DrvPrior        driver process priority
0022   00           dc.b    SBFFlags        file manager flags
0023   00           dc.b    DrivFlag        driver flags
0024 0000           dc.w    DMAMode         DMA type/usage
 0000000e OptLen    equ     *-OptTbl

0026 5342 FileMgr   dc.b    "SBF",0         Sequential block file manager
```

SBF Descriptor Example: (continued)

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/06  14:53  Page     2
 mt0_tm3000.a
MT0 Device Descriptor - Device Descriptor for Tape controller


          SBFDesc     macro

          Port        equ       \1              Port address
          Vector      equ       \2              autovector number
          IRQLevel    equ       \3              hardware interrupt level
          Priority    equ       \4              polling priority
          DevDrv      dc.b      "\5",0          driver module name
                      ifgt      \#-5            standard device setup requested?


                      endc
                      endm

***************
* Descriptor Defaults
 00000067 Mode        set       Share_+ISize_+Exec_+Updat_
 00000000 DevCon      set       0
 00000000 Speed       set       0               driver defined
 00000002 NumBlks     set       2
 00002000 BlkSize     set       0x2000
 00000100 DrvPrior    set       256
 00000000 SBFFlags    set       0
 00000000 DrivFlag    set       0
 00000000 DMAMode     set       0               driver defined



 00000000 DrvNum      set       0
 00000004 NumBlks     set       4
 00004000 BlkSize     set       0x4000
 000003e8 DrvPrior    set       1000
                      SBFDesc   0xFF1100,29,5,128,"tm3000"


 00000032             ends

Errors: 00000
Memory used: 23k
Elapsed time: 7 second(s)
```

EXAMPLES

INIT Module Example:

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/04  09:42  Page  1  init.a
Init: OS-9 Configuration Module -
00001                          nam       Init:           OS-9 Configuration Module
00002
00019  00000009 Edition       equ       9               current edition number
00020
00021  00000c00 Typ_Lang      set       (Systm<<8)+0
00022  00008000 Attr_Rev      set       (ReEnt<<8)+0
00023                          psect     init,Typ_Lang,Attr_Rev,Edition,0,0
00024
00025 * Configuration constants (default; changable in "systype.d" file)
00026  000109a0 CPUTyp        set       68000           cpu type (68008/68000/68010)
00027  00000001 Level         set       1               OS-9 Level One
00028  00000002 Vers          set       2               Version 2.0
00029  00000000 Revis         set       0
00030  00000000 Edit          set       0               Edition
00031  00000000 Site          set       0               installation site code
00032  00000080 MDirSz        set       128             initial mod dir size (unused)
00033  00000020 PollSz        set       32              IRQ polling table size (fixed)
00034  00000020 DevCnt        set       32              device table size (fixed)
00035  00000040 Procs         set       64              init proc table size (div by 64)
00036  00000040 Paths         set       64              init path table size (div by 64)
00037  00000002 Slice         set       2               ticks per time slice
00038  00000080 SysPri        set       128             initial system priority
00039  00000000 MinPty        set       0               initial sys min executable prior
00040  00000000 MaxAge        set       0               initial sys max natural age limit
00041  00000000 MaxMem        set       0               top of RAM (unused)
00042  00000000 Events        set       0               init event table size (div by 8)
00043  00000000 Compat        set       0               compat ver (lsb=1: IRQ save regs)
00044                          use       defsfile        (defsfile overrides above defs)
00045
00046 * Configuration module body
00047 0000 0000               dc.l      MaxMem          (unused)
00048 0004 0020               dc.w      PollSz          IRQ polling table size
00049 0006 0020               dc.w      DevCnt          device table size
00050 0008 0040               dc.w      Procs           initial process table size
00051 000a 0040               dc.w      Paths           initial path table size
00052 000c 0088               dc.w      SysParam        param string for first exec mod
00053 000e 007d               dc.w      SysStart        first exec mod name offset
00054 0010 009c               dc.w      SysDev          system default device name offset
00055 0012 00a0               dc.w      ConsolNm        standard I/O pathlist name offset
00056 0014 0064               dc.w      Extens          Customization module name offset
00057 0016 00a6               dc.w      ClockNm         clock module name offset
00058 0018 0002               dc.w      Slice           number of ticks per time slice
00059 001a 006a               dc.w      UsrAct          accounting module name offset
00060 001c 0000               dc.l      Site            installation site code
00061 0020 0070               dc.w      MainFram        installation name offset
00062 0022 0001               dc.l      CPUTyp          specific 68000 family processor
```

INIT Module Example: (continued)

```
00063 0026 0102             dc.b    Level,Vers,Revis,Edit OS-9 Level
00064 002a 0054             dc.w    OS9Rev          OS-9 revision string offset
00065 002c 0080             dc.w    SysPri          initial system priority
00066 002e 0000             dc.w    MinPty          initial sys min exec prior
00067 0030 0000             dc.w    MaxAge          maximum system natural age limit
00068 0032 0000             dc.l    MDirSz          module directory size (unused)
00069 0036 0000             dc.w    Events          init ev tab size (num of entries)
00070 0038   00             dc.b    Compat          version change smooth byte
00071 0039   00             dc.b    0
00072 003a 0000             dc.w    0,0,0,0,0,0     reserved
00073 0046 0000             dc.w    0,0,0,0,0,0,0   reserved
00074
00075 * Configuration name strings
00076 0054 4f53 OS9Rev       dc.b    "OS-9/"
00080 0059 3638             dc.b    "68000"
00085 005e 2056             dc.b    " V",Vers+'0',".",Revis+'0',0
00086 0064 4f53 Extens      dc.b    "OS9P2",0       Customization module name
00087 006a 5541 UsrAct      dc.b    "UAcct",0       user accounting module name
00088
00089 * The remaining names are defined in the "systype" macro below
00090                        CONFIG
00091  000000ae              ends
00092
Errors: 00000
Memory used: 31k
Elapsed time: 22 second(s)
```

EXAMPLES

SYSGO Module Example:

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/04  09:43  Page 1  sysgo.a
Sysgo - OS-9/68000 Initial (startup) module
00001                          nam       Sysgo
00002                          ttl       OS-9/68000      Initial (startup) module
00003
00015  00000004 Edition        equ       4               current edition number
00016
00017  00000101 Typ_Lang       set       (Prgrm<<8)+Objct
00018  00000000 Attr_Rev       set       0               (non-reentrant)
00019                          psect     sysgo,Typ_Lang,Attr_Rev,Edition,0,Entry
00020
00021                          use       defsfile
00022
00023                          vsect
00024  00000000                ds.b      255             stack space
00025  00000000                ends
00026
00027 0000=4e40 Intercpt       os9       F$RTE           return from intercept
00028
00029 0004 41fa Entry          lea       Intercpt(pc),a0
00030 0008=4e40                os9       F$Icpt
00031 000c 41fa                lea       CmdStr(pc),a0   default execution directory ptr
00032 0010 7004                moveq     #Exec_,d0       execution mode
00033 0012=4e40                os9       I$ChgDir        change exec dir (ignore errors)
00034 0016 640c                bcc.s     Entry10         continue if no error
00035 0018 7001                moveq     #1,d0           std output path
00036 001a 721a                moveq     #ChdErrSz,d1    size
00037 001c 41fa                lea       ChdErrMs(pc),a0 "Help, I can't find CMDS"
00038 0020=4e40                os9       I$WritLn        output error message
00039
00040 * Process startup file
00041 0024 7000 Entry10        moveq     #0,d0           std input path
00042 0026=4e40                os9       I$Dup           clone it
00043 002a 3e00                move.w    d0,d7           save cloned path number
00044 002c 7000                moveq     #0,d0           std input path
00045 002e=4e40                os9       I$Close
00046 0032 303c                move.w    #Read_,d0
00047 0036 41fa                lea       Startup(pcr),a0 "startup" pathlist
00048 003a=4e40                os9       I$Open          open startup file
00049 003e 640e                bcc.s     Entry15         continue if no error
00050 0040 7001                moveq     #1,d0           std output path
00051 0042 7220                moveq     #StarErSz,d1    size of startup error msg
00052 0044 41fa                lea       StarErMs(pc),a0 "Help, I can't find 'startup'"
00053 0048=4e40                os9       I$WritLn        output error message
00054 004c 6032                bra.s     Entry25
00055
```

SYSGO Module Example: (continued)

```
00056 004e 7000 Entry15    moveq    #0,d0           any type module
00057 0050 7200            moveq    #0,d1           no additional default memory size
00058 0052 7406            moveq    #StartPSz,d2    size of startup shell params
00059 0054 7603            moveq    #3,d3           copy three std I/O paths
00060 0056 7800            moveq    #0,d4           same priority
00061 0058 41fa            lea      ShellStr(pcr),a0 shell name
00062 005c 43fa            lea      StartPrm(pcr),a1 initial parameters
00063 0060=4e40            os9      F$Fork          fork shell
00064 0064 6410            bcc.s    Entry20         continue if no error
00065 0066 7001            moveq    #1,d0           std output path
00066 0068 7219            moveq    #FrkErrSz,d1    size
00067 006a 41fa            lea      FrkErrMs(pc),a0 "oh dear, can't fork Shell"
00068 006e=4e40            os9      I$WritLn        output error message
00069 0072=4e40            os9      F$SysDbg        crash system
00070
00071 0076=4e40 Entry20    os9      F$Wait          wait for death, ignore any error
00072 007a 7000            moveq    #0,d0           std input path
00073 007c=4e40            os9      I$Close         close redirected "startup"
00074 0080 3007 Entry25    move.w   d7,d0
00075 0082=4e40            os9      I$Dup           restore original std input
00076 0086 3007            move.w   d7,d0
00077 0088=4e40            os9      I$Close         remove cloned path
00078
00079 008c 7000 Loop       moveq    #0,d0           any type module
00080 008e 7200            moveq    #0,d1           default memory size
00081 0090 7401            moveq    #1,d2           one parameter byte (CR)
00082 0092 7603            moveq    #3,d3           copy std I/O paths
00083 0094 7800            moveq    #0,d4           same priority
00084 0096 41fa            lea      ShellStr(pcr),a0 shell name
00085 009a 43fa            lea      CRChar(pcr),a1 null paramter string
00086 009e=4e40            os9      F$Fork          fork shell
00087 00a2 650a            bcs.s    ForkErr         abort if error
00088 00a4=4e40            os9      F$Wait          wait for it to die
00089 00a8 6504            bcs.s    ForkErr
00090 00aa 4a41            tst.w    d1              zero status?
00091 00ac 67de            beq.s    Loop            loop if so
00092 00ae=4e40 ForkErr    os9      F$PErr          print error message
00093 00b2 60d8            bra.s    Loop
00094
00095 00b4 7368 ShellStr   dc.b     "shell",0
00096 00ba=5379 FrkErrMs   dc.b     "Sysgo can't fork 'shell'",C$CR
00097  00000019 FrkErrSz   equ      *-FrkErrMs
00098
00099 00d3 434d CmdStr     dc.b     "CMDS",0
00100 00d8=5379 ChdErrMs   dc.b     "Sysgo can't chx to 'CMDS'",C$CR
00101  0000001a ChdErrSz   equ      *-ChdErrMs
00102
```

SYSGO Module Example: (continued)

```
00103 00f2 7374 Startup    dc.b     "startup",0
00104 00fa=5379 StarErMs   dc.b     "Sysgo can't open 'startup' file",C$CR
00105  00000020 StarErSz   equ      *-StarErMs
00106
00107 011a 2d6e StartPrm   dc.b     "-npxt"
00108 011f=  00 CRChar     dc.b     C$CR
00109  00000006 StartPSz   equ      *-StartPrm
00110  00000120            ends
00111
Errors: 00000
Memory used: 31k
Elapsed time: 21 second(s)
```

End Of Appendix B

NOTES

ERROR CODES

ERROR NUMBER	DESCRIPTION
000:200 E$PthFul	PATH TABLE FULL - This error is returned when a user program has tried to open more than 32 I/O paths simultaneously. When the system path table becomes full, the kernel automatically expands it. However, this error could be returned if there is not enough (contiguous) memory to expand it.
000:201 E$BPNum	ILLEGAL PATH NUMBER - This error is returned when the path number was too large or for a non-existent path. This could occur whenever passing a path number to an I/O call.
000:202 E$Poll	INTERRUPT POLLING TABLE FULL - This error is returned when an attempt is made to install an IRQ Service Routine into the system polling table, and the table is full. To install another interrupt producing device, one must first be removed. The system's INIT module specifies the maximum number of IRQ devices that may be installed.
000:203 E$BMode	ILLEGAL MODE - This error is returned when an attempt is made to perform an I/O function of which the device or file was incapable. This could occur, for instance, when trying to read from an output file (for example, a printer).
000:204 E$DevOvf	DEVICE TABLE FULL - This error is returned when the specified device can not be added to the system because the device table is full. To install another device, one must first be removed. The system's INIT module specifies the maximum number of devices that may be supported, but this may be changed to add more.
000:205 E$BMID	ILLEGAL MODULE HEADER - This error is returned when the specified module can not be loaded because its module sync code is incorrect.

ERROR NUMBER	DESCRIPTION
000:206 E$DirFul	MODULE DIRECTORY FULL - This error is returned when the specified module can not be added to the system because the module directory is full. To load or create another module, one must first be unlinked. Although OS-9 expands the module directory when it becomes full, this error may be returned because there is not enough memory or the memory is too fragmented to use.
000:207 E$MemFul	MEMORY FULL - This error is returned when the process will not execute because there is not enough contiguous RAM free. This can also occur if a process has already been allocated the maximum number of blocks permitted by the system.
000:208 E$UnkSvc	ILLEGAL SERVICE REQUEST - This error is returned when the specified service call has an unknown or invalid service code number. This can also occur if a getstat/ setstat call is made with an unknown status code.
000:209 E$ModBsy	MODULE BUSY - This error is returned when trying to access a non-sharable module that is in use by another process.
000:210 E$BPAddr	BOUNDARY ERROR - This error is returned when a memory deallocation request is not passed a valid block address or an attempt is made to deallocate memory not previously assigned.
000:211 E$EOF	END OF FILE - This error is returned when an end of file condition is encountered on a read operation.
000:212 E$VctBsy	VECTOR BUSY - This error is returned when a device is trying to use an IRQ vector that is currently being used by another device.

ERROR NUMBER	DESCRIPTION
000:213 E$NES	NON-EXISTING SEGMENT - This error is returned when a search is made for a disk file segment that could not be found. The device may have a damaged file structure.
000:214 E$FNA	FILE NOT ACCESSIBLE - This error is returned when trying to open a file or device without the correct access permissions. Check the file's attributes and the owner ID.
000:215 E$BPNam	BAD PATH NAME - This error is returned when there is a syntax error in the specified pathlist (illegal character, etc.). This can occur whenever referencing a path by name.
000:216 E$PNNF	PATH NAME NOT FOUND - This error is returned when the specified pathlist can not be found. This could be caused by misspellings or incorrect directories, etc.
000:217 E$SLF	SEGMENT LIST FULL - This is returned when a file is too fragmented to be expanded any further. This can be caused by expanding a file many times without regard to allocation of memory. It also occurs on disks with little free memory or disks whose free memory is too scattered. A simple way to solve this problem is to copy the file (or disk). This should move it into more contiguous areas.
000:218 E$CEF	FILE ALREADY EXISTS - This error occurs when trying to create a file using a name that already appears in the current directory.
000:219 E$IBA	ILLEGAL BLOCK ADDRESS - This error is returned when a search for an illegal block address has occurred. An invalid pointer or block size has been passed or the device's file structure is damaged.

ERROR NUMBER	DESCRIPTION
000:220 E$HangUp	TELEPHONE (MODEM) DATA CARRIER LOST
000:221 E$MNF	MODULE NOT FOUND - This error is returned when a request is made to link to a module that is not found in the module directory. Modules whose headers have been modified or corrupted will not be found.
000:222 E$NoClk	NO CLOCK - This error is returned when a request is made that uses the system clock and the system has no clock running. For example, a timed SLEEP request will return this error if there is no system clock running. SETIME is used to start the system clock.
000:223 E$DelSP	SUICIDE ATTEMPT - This error is returned when a User requests to deallocate and return the memory where the user's stack is located. This could be caused, for example, by using the F$Mem system call to contract the data memory of the specified process.
000:224 E$IPrcID	ILLEGAL PROCESS NUMBER - This error is returned when a system call is passed a process ID to a non-existent process or a process that the user may not access.
000:225 E$Param	BAD POLLING PARAMETER - This error is returned when an impossible vector number is passed to the IRQ polling system.
000:226 E$NoChld	NO CHILDREN - This error is returned when a F$Wait request is made and the process has no child process for which to wait.
000:227 E$ITrap	ILLEGAL TRAP CODE - This error is returned when an unavailable (already in use) or invalid trap code is used in a TLINK call.

ERROR NUMBER	DESCRIPTION
000:228 E$PrcAbt	PROCESS ABORTED - This error is returned when a process is aborted by the kill signal code.
000:229 E$PrcFul	PROCESS TABLE FULL - This error is returned when the system process table is full (too many processes currently running). Although OS-9 automatically tries to expand the table, this error may occur if there is not enough contiguous memory to do so.
000:230 E$IForkP	ILLEGAL PARAMETER AREA - This error occurs when ridiculous parameters are passed to fork call.
000:231 E$KwnMod	KNOWN MODULE - This error is returned when a call is made to install a module that is already in memory.
000:232 E$BMCRC	INCORRECT MODULE CRC - This error is returned when the specified module being checked or verified has a bad CRC value. To generate a valid CRC, use the FIXMOD utility.
000:233 E$USigP	UNPROCESSED SIGNAL PENDING
000:234 E$NEMod	NON-EXECUTABLE MODULE - This error is returned when a process tries to execute a module with a type other than program/object.
000:235 E$BNam	BAD NAME - This error occurs when there is a syntax error in the specified name.

ERROR NUMBER	DESCRIPTION
000:236 E$BMHP	BAD PARITY - This error is returned when the specified module has bad module header parity.
000:237 E$NoRAM	RAM FULL - This error occurs when there is no free system RAM available at the time of the request for memory allocation. This also occurs when there is not enough contiguous memory to process a fork request.
000:238 E$DNE	DIRECTORY NOT EMPTY - This error is returned when attempting to remove the directory attribute from a directory that is not empty.
000:239 E$NoTask	NO TASK NUMBER AVAILABLE - This error occurs when all task numbers are currently in use and a request is made for execution or creation of a new task.

DEVICE DRIVER ERRORS

The following error codes are generated by I/O device drivers, and are somewhat hardware dependent.

ERROR NUMBER	DESCRIPTION
000:240 E$Unit	ILLEGAL DRIVE NUMBER
000:241 E$Sect	BAD SECTOR - bad disk sector number.
000:242 E$WP	WRITE PROTECT - device is write protected.
000:243 E$CRC	CRC ERROR - CRC error on read or write verify.
000:244 E$Read	READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
000:245 E$Write	WRITE ERROR - hardware error during disk write operation.
000:246 E$NotRdy	NOT READY - device has "not ready" status.
000:247 E$Seek	SEEK ERROR - physical seek to non-existent sector.
000:248 E$Full	MEDIA FULL - insufficient free space on media.
000:249 E$BTyp	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive).
000:250 E$DevBsy	DEVICE BUSY - non-sharable device is in use.

ERROR NUMBER	DESCRIPTION
000:251 E$DIDC	DISK ID CHANGE - This error is returned when the disk media was changed with open files. RBF copies the disk ID number (from sector 0) into the path descriptor of each path when it is opened. If this does not agree with the driver's current disk ID, this error is returned. The driver updates the current disk ID only when sector 0 is read. It is thus possible to swap disks without RBF noticing. This check helps to prevent this possibility.
000:252 E$Lock	RECORD IS LOCKED-OUT - This error is returned if another process is accessing the requested record. Normal record locking routines will wait forever for a record in use by another user to become available. However, RBF may be told to wait for a finite amount of time with a setstat. If the time expires before the record becomes free, this error is returned.
000:253 E$Share	NON-SHARABLE FILE BUSY - The requested file or device has the single user bit set or was opened in single user mode and another process is accessing the requested file. A common way to get this error is to attempt to delete a file that is currently open.
000:254 E$DeadLk	I/O DEADLOCK - Two processes are attempting to use the same two disk areas simultaneously. Each process is locking out the other process, producing the I/O deadlock. To proceed, one of the two processes must release its control to allow the other to proceed.
000:255 E$Format	DEVICE IS FORMAT PROTECTED - This error is returned when an attempt is made to format a disk that has been format protected. A bit in the device descriptor may be changed to allow the device to be formatted. Formatting is usually inhibited on hard disks to prevent erasure.

PROCESSOR EXCEPTION ERROR CODES

Error numbers in the range 100-154 are reserved to indicate that a processor related exception occurred on behalf of the program. Only exceptions represented by the error numbers given below can occur on behalf of the user program. Other error numbers in the range of 100-154 are reserved. Unless the program arranges for special handling of the exception condition (F$STrap), the error is considered fatal and the program is terminated. The following error numbers represent the hardware exception vector number plus 100.

ERROR NUMBER	DESCRIPTION
000:102 E$BusErr	BUS ERROR - bus error exception occurred.
000:103 E$AdrErr	ADDRESS ERROR - address error exception occurred.
000:104 E$IllIns	ILLEGAL INSTRUCTION - illegal instruction exception occurred.
000:105 E$ZerDiv	ZERO DIVIDE - zero divide exception occurred.
000:106 E$Chk	CHECK -CHK instruction exception occurred.
000:107 E$TrapV	TRAPV - TrapV instruction exception occurred.
000:108 E$Violat	PRIVILEGE VIOLATION - privilege violation exception occurred.
000:109 E$Trace	UNINITIALIZED TRACE EXCEPTION - an uninitialiazed trace exception occurred.
000:110 E$1010	1010 TRAP - A Line emulator exception occurred.
000:111 E$1111	1111 TRAP - F Line emulator exception occurred.

ERROR NUMBER	DESCRIPTION
000:113	COPROCESSOR PROTOCOL VIOLATION
000:114	FORMAT ERROR
000:115	UNINITIALIZED INTERRUPT OCCURRED
000:124	SPURIOUS INTERRUPT OCCURRED
000:133 - 000:147 E$Trap	uninitialized user TRAP 1-15 executed.
000:148 E$FPUnordC	FPCP ERROR - Branch or set on unordered condition error.
000:149 E$FPInxact	FPCP ERROR - Inexact result.
000:150 E$FPDivZer	FPCP ERROR - Divide by zero error.
000:151 E$FPUndrFl	FPCP ERROR - Underflow error.
000:152 E$FPOprErr	FPCP ERROR - Operand error.
000:153 E$FPOverFl	FPCP ERROR - Overflow error.
000:154 E$FPNotNum	FPCP ERROR - NAN signaled.
000:156	PMMU CONFIGURATION ERROR

ERROR NUMBER	DESCRIPTION
000:157	PMMU ILLEGAL OPERATION
000:158	PMMU ACCESS LEVEL VIOLATION

OTHER ERRORS

000:002	KEYBOARD QUIT - This error is returned when the "keyboard abort" function (control E) is sent.
000:003	KEYBOARD INTERRUPT - This error is returned when the "keyboard interrupt" function (control C) is sent.
000:064 E$IllFnc	ILLEGAL FUNCTION CODE - Math trap handler error.
000:065 E$FmtErr	FORMAT ERROR - Math trap handler error.
000:066 E$NotNum	NUMBER NOT FOUND - Math trap handler error.
000:067 E$IllArg	ILLEGAL ARGUMENT - Math trap handler error.
000:164 E$Permit	NO PERMISSION - The process or module must be owned by the super user to perform the requested function.
000:165 E$Differ	DIFFERENT ARGUMENTS - F$ChkNam arguments do not match.
000:166 E$StkOvf	STACK OVERFLOW - F$ChkNam can cause this error if the pattern string is too complex.

ERROR NUMBER	DESCRIPTION
000:167 E$EvntID	ILLEGAL EVENT ID - This error is returned when an invalid or illegal event ID number is specified.
000:168 E$EvNF	EVENT NAME NOT FOUND - This error is returned when an attempt to link to or delete an event is made, but the name is not found in the event table.
000:169 E$EvBusy	EVENT BUSY - This error is returned when an attempt to delete an event is made and its link count is non-zero. This can also occur if an attempt to create an already existent named event is made.
000:170 E$EvParm	IMPOSSIBLE EVENT PARAMETER - This error is returned when impossible parameters are passed to F$Event.
000:171 E$Damage	SYSTEM DAMAGE - A system data structure has been corrupted.
000:172 E$BadRev	INCOMPATIBLE REVISION - Software revision is incompatible with operating system revision.
000:173 E$PthLost	PATH LOST - Path became lost because network node was down.
000:174 E$BadPart	BAD PARTITION - Bad partition data or no active partition.

End Of Appendix C

INDEX

For an alphabetical listing of the OS-9 system calls, see Appendix A.

A

Active state 4-3
Allocation Map 7-2, 7-3
Auto-Vectored Interrupts 4-8

D

DEFS Files 12-1, 12-2
  Sys.l 3-2, 12-1
  Usr.l 3-2, 12-1
Device Descriptors
  Module header 1-7, 1-8, 6-4, 6-5
  NFM Initialization Table 11-15 through 1-18
  Overview i, ii, 6-4
  RBF Initialization Table 7-1, 7-12, 7-13, 7-14
  SBF Initialization Table 9-2, 9-3
  SCF Initialization Table 8-1 through 8-6
  Segment allocation (RBF) 7-5
Device Drivers
  Functions 6-2
  Format 6-2
  Interrupts 6-3
  Overview i, ii, 6-1
  NFM Drivers
    Message Handling 11-13, 11-14
    Storage allocation 11-22
    Subroutines
      GETSTAT 11-29
      INIT 11-24
      READ 11-25
      SETSTAT 11-29
      TERM 11-30
      IRQ 11-31
      WRITE 11-27
  RBF Drivers
    Driver Table 7-21
    Storage allocation 7-18, 7-19, 7-20

D (continued)

Device Drivers
  RBF (continued)
    Subroutines
      GETSTAT 7-26
      INIT 7-23
      READ 7-24
      SETSTAT 7-26
      TERM 7-27
      TRAP 7-28
      WRITE 7-25
    Requirements 6-1
  SBF Drivers
    Driver Tables 9-6, 9-8
    Overview 9-5
    Storage allocation 9-5, 9-6, 9-7
    Subroutines
      GETSTAT 9-13
      INIT 9-10
      READ 9-11
      SETSTAT 9-13
      TERM 9-14
      IRQ 9-15
      WRITE 9-12
  SCF Drivers
    Overview 8-9
    Storage allocation 8-10, 8-11, 8-12
    Subroutines
      GETSTAT 8-17
      INIT 8-14
      READ 8-15
      SETSTAT 8-17
      TERM 8-18
      TRAP 8-19
      WRITE 8-16
  Static Storage 6-2

E

Error Exceptions 4-8
Events
  Creating 15-5
  Definition 15-1

E (co d)

Events (continued)
F$Event 15-3
Ev$Creat 15-5
Ev$Delet 15-5
Ev$Info 15-7
Ev$Link 15-4
Ev$Pulse 15-8
Ev$Read 15-7
Ev$Set 15-9
Ev$SetR 15-9
Ev$Signl 15-8
Ev$UnLnk 15-4
Ev$Wait 15-6
Ev$WaitR 15-6
Exception processing 4-6 through 4-9

F

File Descriptors 7-4
File Managers
Function 5-3, 5-4, 5-5
Module header 1-7, 1-8, 1-9
NFM Chapter 11
Organization 5-2
Overview i, ii, 5-1
Pipeman Chapter 10
RBF Chapter 7
SBF Chapter 9
SCF Chapter 8

I-L

Identification sector 7-2
INIT module 2-1 through 2-4, Apdx B
Interrupt Processing 4-6 through 4-9, 6-3
I/O System Calls 3-3, 3-4, Chapter 17

Kernel
Execution Scheduling 4-5, 4-6
Exception Processing 4-6, through 4-9
Overview i, ii, 1-1
Memory Management 1-1
System Call Processing 3-3

Link count 1-2, 1-3
Logical Sector Number 7-1

M

Math Module Chapter 12
Calling 14-1
Data Format 14-2
Memory Map
Allocation 1-12, 1-13, 1-14
Operating System Object Code 1-13
System Dynamic Memory 1-13
System Global Memory 1-13
User Memory 1-14
Fragmentation 1-14
Typical OS-9 Map 1-14
Memory Modules
CRC 1-2, 1-11
Header 1-2, 1-4 through 1-10
Module directory 1-2
Requirements 1-3
ROMed Module 1-11
Structure 1-2
Types 1-3
Multitasking 4-1

N

Networking
Broadcasting 11-4
Device Descriptor 11-15 through 11-18
Device Driver 11-13, 11-14, 11-22
through 11-31
Errors 11-8, 11-9
Messages 11-14
NFM 11-10 11-11, 11-12
Node Name Module 11-20
Overview 11-1, 11-2, 11-3
Path Descriptor 11-18
Requirements 11-1, 11-2
Security 11-5, 11-6, 11-7

P

Path Descriptors
NFM option table 11-19
Overview 6-7, 6-8
Pipeman option table 10-6
RBF option table 7-17
SBF option table 9-4
SCF option table 8-8

P (continued)

Pipeman
- Overview 10-1
- Path Descriptor Option Table 10-6
- Pipes
  - Closing 10-4
  - Creating 10-2
  - Directories 10-5
  - Named 10-1
  - Reading 10-3
  - Status 10-4, 10-5
  - Unnamed 10-1
  - Writing 10-3

Primary Module 4-2, 4-4
Process
- Age 4-5
  - Minimum priority 4-6
  - Maximum age 4-6
- Creation 4-2
- Descriptor 4-2
- Execution Scheduling 4-3, 4-5
  - Timeslicing 4-1
- ID 4-3
- Initialization 4-3
- Memory Areas 4-2, 4-3, 4-4
- Priority 4-5, 4-6
- States 4-3
- Syncronization 15-1, 15-2
- Termination 4-3

Program module header 1-7, 1-8, 1-9, 1-10

R

RBF
- Device Descriptor
  - Initialization Table 7-11 through 7-15
- Disk Organization
  - Allocation Map 7-2
  - Identification sector 7-2,
  - Logical Sector Number 7-1
  - Root Directory 7-2, 7-3
- Drivers
  - Driver Table 7-21
  - Storage allocation 7-18, 7-19, 7-20

RBF (continued)
- Driver Subroutines
  - GETSTAT 7-26
  - INIT 7-23
  - READ 7-24
  - SETSTAT 7-26
  - TERM 7-27
  - TRAP 7-28
  - WRITE 7-25
- File Organization
  - Directory file format 7-5, 7-6
  - File descriptor 7-4
  - Segment allocation 7-4
- Overview 7-1
- Raw I/O 7-6
- Record locking 7-6 through 7-10

Record Locking 7-7 through 7-10
- End-of-file lock 7-8
- File lock 7-7
- Dead lock 7-8

Re-entrant code 1-3
Re-entrant modules 1-3
Reset Vectors 4-7

S

SBF
- Device descriptor 9-2, 9-3
- Drivers
  - Overview 9-5
  - Storage
  - Subroutines
    - GETSTAT 9-13
    - INIT 9-10
    - READ 9-11
    - SETSTAT 9-13
    - TERM 9-14
    - TRAP 9-15
    - WRITE 9-12
- Driver Tables 9-8
- Path descriptor 9-4
- Tape I/O 9-1, 9-2

S (continued)

SCF
Device descriptor 8-1 through 8-6
Drivers
Overview 8-9
Storage allocation 8-10, 8-11, 8-12
Subroutines
GETSTAT 8-17
INIT 8-14
READ 8-15
SETSTAT 8-17
TERM 8-18
TRAP 8-19
WRITE 8-16
Path descriptor 8-7, 8-8
Segment allocation (RBF) 7-5
Sleeping State 4-3
Step Rate 7-11
Sys.l library file 3-2, 12-1
Sysgo Module 2-5, Apdx B
System Initialization
SEE INIT module
System State Processes 3-1 through 3-4
System State System Calls Chapter 17

T

Tape I/O 9-1, 9-2
Timeslicing 4-1
Trace Exceptions 4-8
Trap Handler
CIO 13-1
Example 13-3 through 13-8
Installing 13-2
Math module 13-1
Module header 1-7, 1-8, 1-9, 1-10
Overview 13-1

U-W

User State Processes 3-1 through 3-4
User State Systemn Calls Chapter 16
User Traps 4-9, 13-1
Usr.l library file 3-2, 12-1

Waiting State 4-3