Microware®

OS-9/68000

OPERATING SYSTEM

TECHNICAL MANUAL

OS-9/68000
OPERATING SYSTEM
TECHNICAL MANUAL

Revision E
Publication date: July 1985
Publication Editor: Walden Miller

Microware Systems Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322
(515)224-1929

PMN - OST68

OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL

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.

PREFACE

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 Basic09. 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.

OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL

TABLE OF CONTENTS

OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL

TABLE OF CONTENTS

SECTION 2: THE INPUT/OUTPUT SYSTEM

TABLE OF CONTENTS

TABLE OF CONTENTS

OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS

OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL

TABLE OF CONTENTS

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, multitasking, 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 and PIPEMAN)

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 (printers or terminals for example). PIPEMAN handles interprocess communication, using memory buffers for data transfer.

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 the "OS-9 OPERATING SYSTEM USER'S MANUAL". 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.

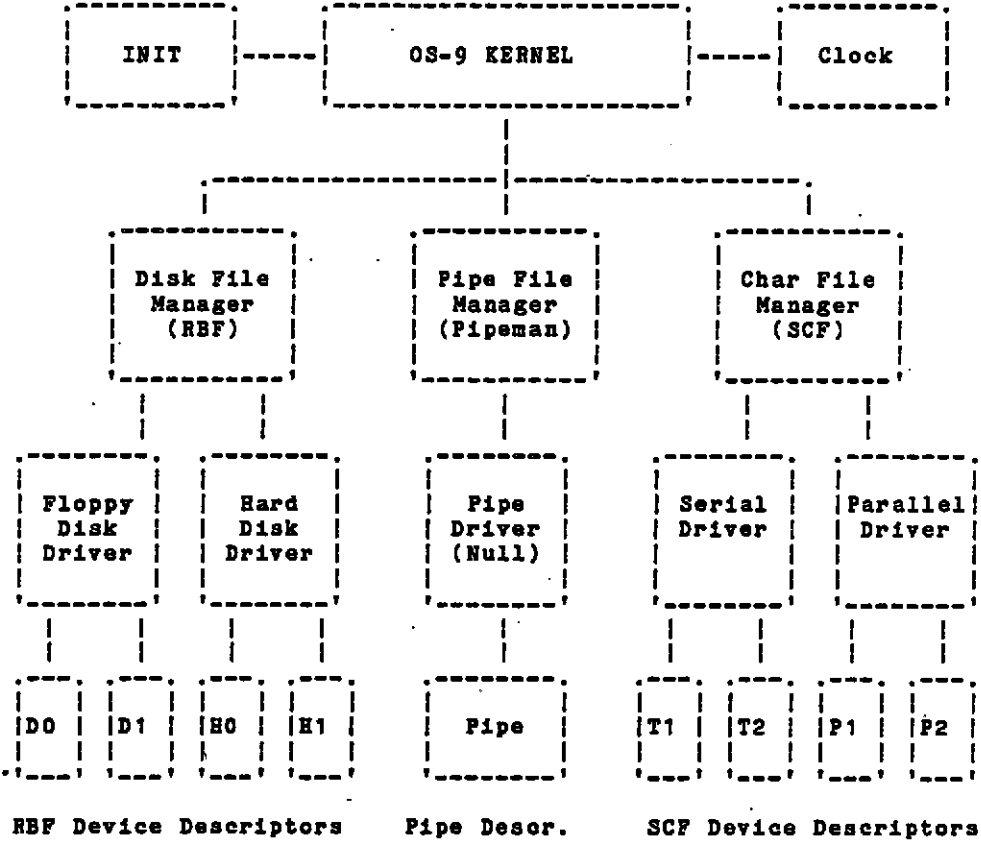


figure 1: OS-9 Component Module Organization

SECTION 1 - THE KERNEL

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

CHAPTER 1
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 14, 15 and 16.

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 convention. 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 execution starting address.

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.

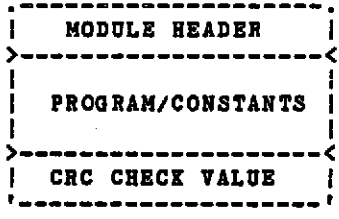


figure 2--Basic Memory Module Format

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 anotner 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 "OS9/68000 MACRO ASSEMBLER 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.

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
$00	M$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 *
$10	M$Acce	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	

figure 3:
Module Header
Standard Fields

* These fields are offsets to strings.

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$Name	Offset to Module Name. The address of the module name string relative to the start (first sync byte) of the module is located here. The name string can be located anywhere in the module and consists of a string of ASCII characters terminated by a null (zero) byte.
M$Acss	Access Permissions. These define the allowable use and access to the module by its owner or by other users. The format and bit patterns are similar to the disk system file security codes. These bytes are reserved for future use.
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

NAME	UTILIZATION

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 = Not Used ("Wild Card" value in system calls)
Object	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.

M$Attr **Attributes**
Bit 7 indicates that the module is re-entrant (sharable by multiple tasks).

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 of a Symbolic debugger.

NAME	UTILIZATION
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.

Additional Header Fields For Individual Modules

Certain types of modules have additional standard header fields following the universal offsets described above. These fields are shown in figure 4 and the table that follows.

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 10. File Manager modules are discussed in Chapters 5, 7 and 8. Device Drivers are discussed in Chapter 6.

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".

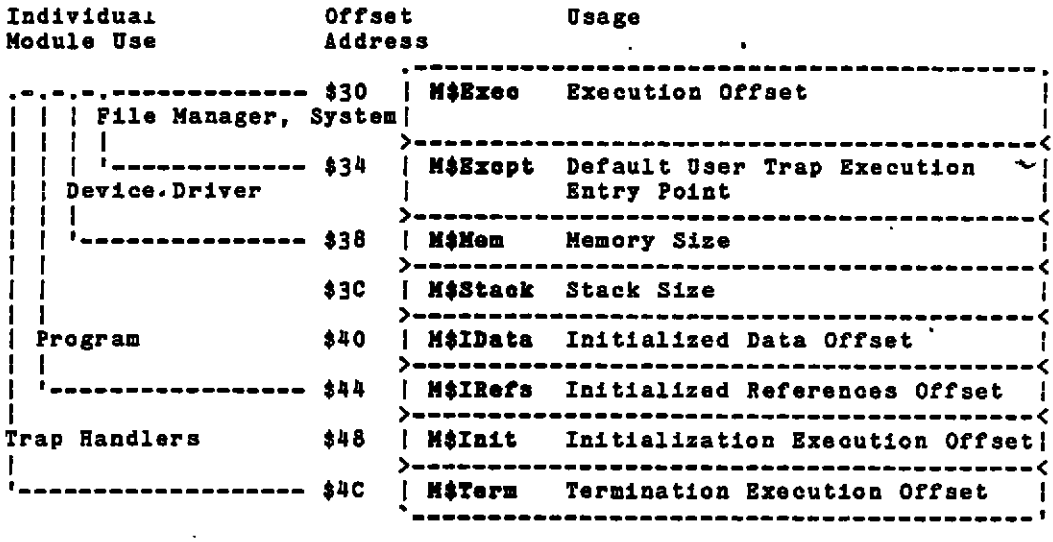


figure 4: Additional Header Fields for Individual Modules

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).

NAME	UTILIZATION

(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.

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:

> The first word of each table is the most significant (MS) word of the offset 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.**

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 linked. 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 Level 1 uses a software memory management system where all memory is contained within a single memory map. Therefore, OS-9 and 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.

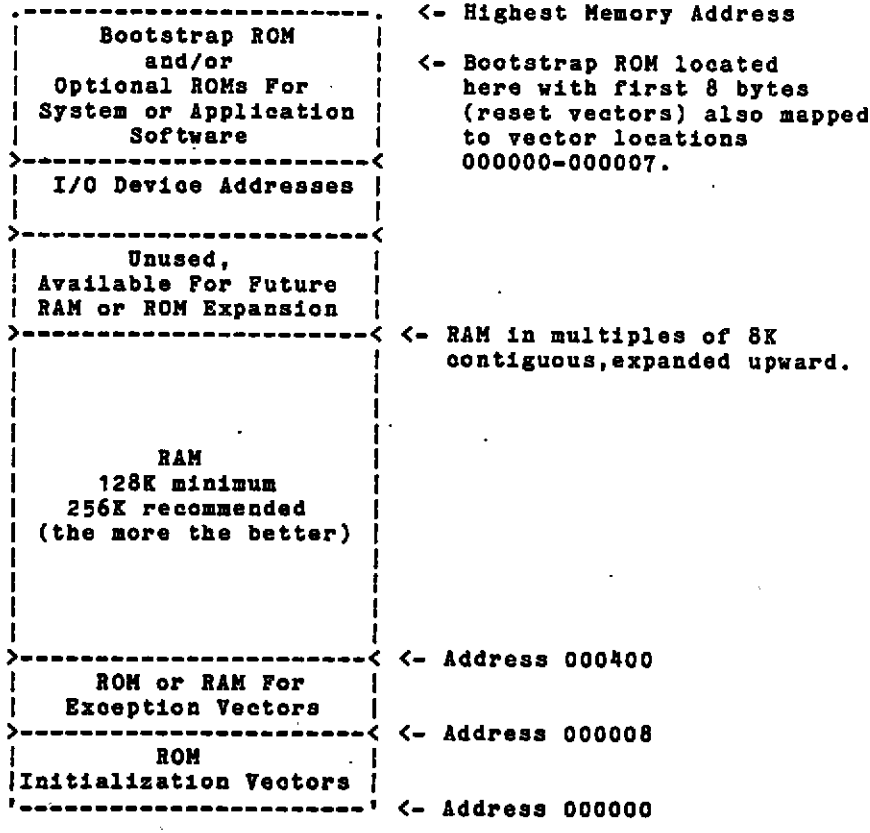


figure 5: Typical OS-9/68000 Memory Map

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 16K 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

USER NOTES

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 table that follows it.

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

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	Reserved	
$3E	M$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	*
$48	M$Slice Ticks per Time Slice	
$4A	M$UsrAct User Accounting Package	*
$4C	Reserved	
$50	M$Instal Installation Name	*
$52	M$CPUTyp CPU Type 68000/68008/68010	
$56	M$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	Reserved (28 bytes)	

figure 6:
Additional fields for the INIT module

* All fields with an "*" are offsets to name strings. The strings themselves follow the 28-byte reserved section.

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$SysGo First executable module name offset.
This is the offset to the name string of the first executable module (usually SYSGO).

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 compliment 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.

NAME	DESCRIPTION
M$Slice	Timeslice. The number of clock ticks per timeslice.
M$UsrAct	User accounting package name offset. This is the offset to the user accounting package name. Microware does not currently supply an accounting package. If supplied by the user, the system will search for it during startup. The default name string to be searched for is "UACCT".
M$Instal	Offset to installation name. This is the offset to the installation name string.
M$CPUTyp	Cpu Type. Cpu type: 68000, 68008, 68010.
M$OS9Lvl	Level, Version and Edition. This four byte field is divided into three parts: level: 1 byte version: 2 bytes edition: 1 byte For example, level 1, version 1.2, edition 0 would be 1120.
M$OS9Rev	Revision offset. This is the offset to the OS-9 level/revision string.
M$SysPri	Priority. This is the system priority that the first module (usually SYSGO) is executed at. This is generally the base priority that all processes start at.
M$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 15 (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 15 (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 14 on Events for discussion of event usage.

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.

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

end of chapter 2

USER NOTES

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 "user.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. Service requests are divided into three categories:

1. **User Mode Function Requests:** 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 Requests:** 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 Mode Function Requests:** These requests are special system calls that can only be used by system software in supervisor 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 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$".

end of chapter 3

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 attempts to load 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 existance or what it contains.

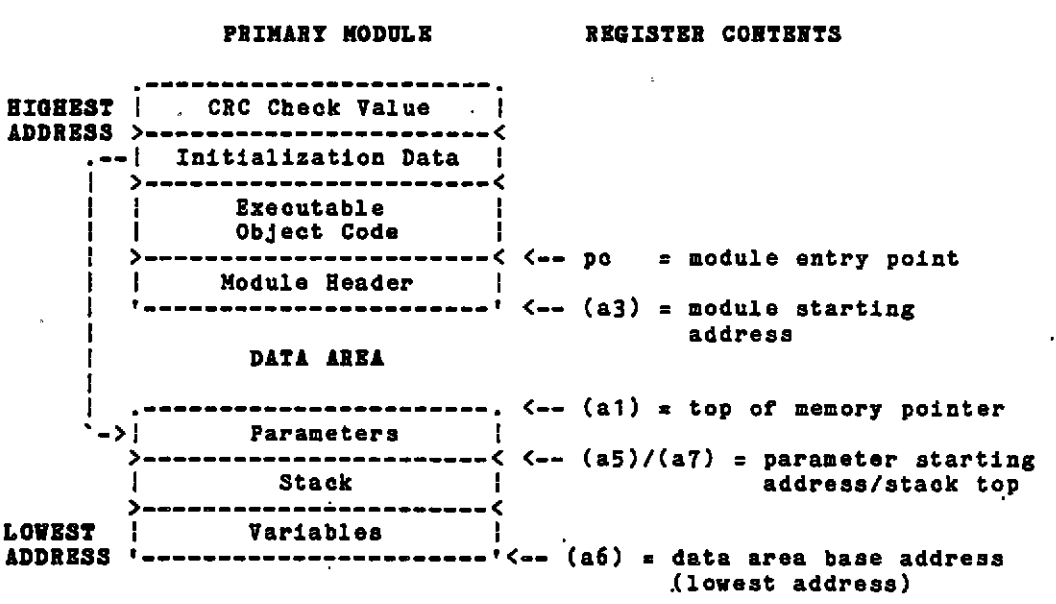
3. **Allocate the Stack and Data Areas.** The primary module's module header contains a data and stack size. OS-9 attempts to allocate a **CONTIGUOUS** memory area of the required size from the free memory space.

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 its identifier. 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.



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		
d3.w	= number of I/O paths inherited	(a4)	= undefined
		(a5)	= parameter pointer
d4.l	= undefined	(a6)	= static storage (data area) base pointer *
d5.l	= parameter size		
d6.l	= total initial memory allocation	(a7)	= stack pointer (same as a5)
d7.l	= undefined		

* (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

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).

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 accessable by the super user 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 important 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. 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 *
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 *

* see section heading "Error Exception"

Vector Number	Related OS-9 Call	Assignment	
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	
48-63	none	(reserved) 16	*
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.

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

SECTION 2 - THE INPUT/OUTPUT SYSTEM

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

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.

The three 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".

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.

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

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

REGISTER	POINTER PASSED
(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

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 should return 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.

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.

Writing past the end of file on a device should expand the file with new data.

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 should perform 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 should call the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing may also be 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 dependence. For example, a SetStat call to format a disk track may behave differently on different types of 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.

end of chapter 5

USER 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.

The name by which the device is known to the system is the name of the device descriptor module. OS-9 copies the information contained in the device descriptor module to the path descriptor data structure for easy access by the drivers.

Device descriptor modules are described in detail later in this chapter.

Driver Module Format

All drivers must conform to the standard OS-9 memory module format. The module type code is "Drivr".

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.

A sample assembly language header is shown in figure 9.

```
* 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

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 Chapter 7 and 8 (for RBF and SCF respectively).

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.

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.

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 outgoing 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 Chapter 7 for the RBF Device Descriptor Module specific table and Chapter 8 for the SCF Device Descriptor Module table).

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 Address		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

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.

NAME	UTILIZATION
M$IRQLvl	IRQ Hardware Interrupt Level.
M$Prior	IRQ Polling Priority. Smaller numbers are polled first if more than one device in 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$Open calls. If a bit is set, the device is capable of performing the corresponding function. The ISize_bit is usually set, because is it usually handled by the file manager or ignored. If the Share_bit (Single User bit) is set here, the device will be non-sharable. This is useful for printers.
M$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 accessable 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.

PATH DESCRIPTORS

Every open path is represented by a data structure called a path descriptor ("PD"). It contains information required by the 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:

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."

figure 11: Universal Path Descriptor Definitions

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

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 SCF and RBF 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.

end of chapter 6

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
$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 vice versa.

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 the extra 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 the super user is 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 method 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 it's 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 encountered. 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.

The super user (group/user ID = 0.0) 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.

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 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	Sectors/Track (default)
$54	PD_T0S	Sectors/Track 0 (default)
$56	PD_SAS	Segment Allocation Size
$58	PD_ILV	Sector Interleave Factor
$59	PD_TFM	DMA Transfer Mode
$5A	PD_TOffs	Track Base Offset
$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

figure 14: Initialization Table for RBF Device Descriptor Modules

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 DiskType device type *
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

* These parameters are format specific.

NAME	UTILIZATION
PD_DN	**Density 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 1-7 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 dependent)

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.**

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.

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	PD_DTP	Device Class
$81	PD_DRV	Drive Number
$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	Disk Write Verification
$8A	PD_SCT	Sectors/Track (default)
$8C	PD_T0S	Sectors/Track 0 (default)
$8E	PD_SAS	Segment Allocation Size
$90	PD_ILV	Sector Interleave Factor
$91	PD_TFM	DMA Transfer Mode
$92	PD_T0ffs	Track Base Offset
$93	PD_S0ffs	Sector Base Offset
$94	PD_SSize	Sector Size (in bytes)
$96	PD_Cntl	Control Word
$98	PD_Trys	# of Tries
$99		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

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

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:

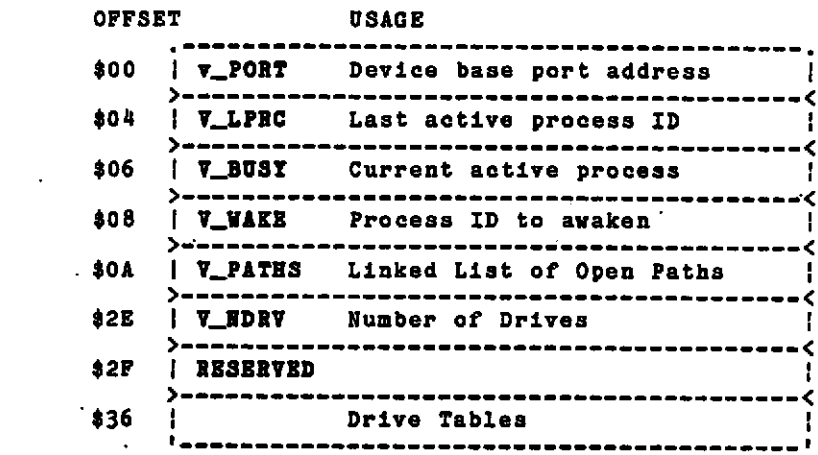


figure 16: Static Storage Allocation

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".

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.

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.
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.

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.

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

The format of each drive table is given in figure 17.

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.

Copied from LSN 0 Identification Sector

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	DD_RES	Reserved
$16	V_TRAK	Current track number
$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 occur
$30	V_HardEr	# of non-recoverable errors
$34	Reserved	(32 bytes)

figure 17:
Device Driver Table Format;

There must be as many tables as were numbered in NDRV.

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 devices 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: d0.1 = number of contiguous sectors to read
d2.1 = 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.

In Version 1.2 of OS-9, RBF only requests one sector reads.

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.

NOTE: On Version 1.2 of OS-9, RBF only requests one sector writes.

NAME: GETSTAT/SETSTAT

INPUT: d0.w = status code
(a1) = address of the device static storage area
(a2) = address of the path descriptor
(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.

end of chapter 7

NAME	UTILIZATION
PD_DTP	Device class (0=SCF 1=RBF 2=PIPE 3=SBF 4=NET)
PD_UPC	Letter case If PD_UPC is unequal to zero, then input or output characters in the range "a..z" are made "A..Z"
PD_BSO	Destructive Backspace When a backspace character is input, SCF will echo PD_BSE (backspace echo character) if PD_BSO is zero. SCF will echo PD_BSE, space, PD_BSE if PD_BSO is non-zero.
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.
PD_DEL	Delete line character This is the input character recognized as the delete line function. See also PD_DLO

NAME	UTILIZATION

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 quit 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 location is normally set to a control-E character.

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

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 re-entrant 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 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 table is shown in figure 18. 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 to change the corresponding control character value in the device descriptor module. Device descriptors may be easily changed using the XMODE utility.

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 "user.l."

OFFSET	USAGE		OFFSET	USAGE	
$48	DV_DTP	Device Type	$56	PD_DUP	Duplicate line character
$49	PD_UPC	Upper case lock	$57	PD_PSC	Pause character
$4A	PD_BSO	Backspace option	$58	PD_INT	Keyboard interrupt character
$4B	PD_DLO	Delete line character	$59	PD_QUT	Keyboard abort character
$4C	PD_ECO	Echo	$5A	PD_BSE	Backspace output
$4D	PD_ALF	Automatic line feed	$5B	PD_OVF	Line overflow character (bell)
$4E	PD_NUL	End of line null count	$5C	PD_PAR	Parity code, # of Stop bits and # of bits per character
$4F	PD_PAU	End of page pause	$5D	PD_BAU	Adjustable baud rate
$50	PD_PAG	Page length	$5E	PD_D2P	Offset to output device name
$51	PD_BSP	Backspace input character	$60	PD_XON	X-ON character
$52	PD_DEL	Delete line character	$61	PD_XOFF	X-OFF character
$53	PD_EOR	End of record character	$62	PD_TAB	Tab character
$54	PD_EOF	End of file character	$63	PD_TABS	Tab column width
$55	PD_RPR	Reprint line character			

figure 18: Device Descripter Initialization Table

NAME	UTILIZATION

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 terminals 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 stop bits as follows:

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

Bits 4 and 5 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 6 and 7 are reserved.

PD_BAU **Software adjustable baud rate**
This one byte field is split into two nibbles. The low order nibble 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	

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.

NAME	UTILIZATION
PD_XON	X-on character See PD_X-off 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$WriteLn 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

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	DV_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	
$91	PD_QUT	Keyboard abort character
$92	PD_BSE	Backspace output
$90	PD_INT	Keyboard interrupt character
$93	PD_OVF	Line overflow character (bell)
$94	PD_PAR	Parity code, # of Stop bits and # of bits per 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."

figure 20: Static Storage Allocation for SCF Device Drivers

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	

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" (Drvr).

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 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
(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) = current process descriptor
(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 indefinately.

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) = current 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
(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
(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 clears 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.

end of chapter 8

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 pipes 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.

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 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 path 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.

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 WriteLn because pipeman does not have to check for carriage returns and the loops moving data are tighter.

Write/WriteLn

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 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.
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 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 you give each pipe descriptor a unique port address (0,1,2,...), 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 21: 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 9

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 derinition 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. A file to be used with the MAKE utility is included to construct the libraries. 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:		Sys.l:	All files in Usr.l plus the following:	
	funcs.a			
	process.a		sysio.a	ioglob.a
	module.a		sysglob.a	iodev.a
	io.a		SCFstat.a	SCFdev.a
	traps.a		RBFstat.a	RBFdev.a
			DRVstat.a	

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 files drvs1.r, drvs2.r, and drvs4.r are relocatable object files that define static storage for RBF and for each disk drive on the system for one, two, or four drives, respectively. Rbfstat.a is merged with one copy of drvstat.a for each drive. So, for a system with two drivers, drvs2.r consists of rbfstat.r and two copies of drvstat.a merged together. The merging is done for you via a file to be used by the MAKE utility. You should link one of the static storage definition files with the relocatable version of your driver.

The "oskdefs.d" file is different than the other DEFS files. The definitions it contains are often in complex expressions that can not be resolved to external references 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:

```
ifp1
use /dd/defs/oskdefs.d
endc
```

end of chapter 10

SECTION 3 - TRAP HANDLERS AND EVENTS

- Trap Handler Modules
- The Math Module
- Events and Semaphores

CHAPTER 11
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: math1 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 math1, refer to chapter 12.

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 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.

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 the program's initialization code. At that time, the particular trap number (1-15) is specified that will to 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$Math1,T$DMul	trap T$Math1 dc.w T$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 re-issued. 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

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
```

EXAMPLE TWO (continued)

```
*********************************
* 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
```

EXAMPLE TRAP HANDLER (continued)

```
*********************************
* Subroutine TrapEnt
* User Trap entry point

* 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.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)
```

EXAMPLE TRAP HANDLER (continued)

```
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

**********************************
* 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
```

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
```

TRACE OF EXAMPLE TWO (continued):

```
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 11

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. Transcedental 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 table on the preceding page. 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$Math1        Trap number of module
dc.w    T$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$Math1,T$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

The math module also provides transcendental and extended mathematical functions. The precision of these routines are 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$Math2,T$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$Math2,T$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$Math2,T$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$Math2,T$AtoD

INPUT: (a0) = Pointer to ascii string
Format: <sign><digits>.<digits><E or e><sign><digits>

OUTPUT: (a0) = Updated pointer
d0:d1 = 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 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$Math2,T$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$Math2,T$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$Math2,T$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$Math2,T$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$Math1,T$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$Math1,T$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$Math1,T$DCmp

INPUT: d0:d1 = First operand
d2:d3 = Second operand

OUTPUT: d0.1 through d3.1 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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$DNrm

INPUT: d0:d1 = 64-bit Unsigned Integer
d2.l = 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$Math1,T$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$Math1,T$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$Math1,T$DtoF

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0.l = 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$Math1,T$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$Math1,T$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$Math1,T$DTrn

INPUT: d0:d1 = Double-precision floating-point number

OUTPUT: d0:d1 = Truncated 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 truncate the fractional part. For example if the number passed is 283.75 this function will return 283.00

T$Exp — Exponential Function — T$Exp

ASSEMBLER CALL: TCALL T$Math1,T$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 = 2.718284 and x is the argument passed).

T$FAdd Single Precision Addition T$FAdd

ASSEMBLER CALL: TCALL T$Math1,T$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$Math1,T$FCmp

INPUT: d0.1 = First operand
d1.1 = Second operand

OUTPUT: d0.1 and d1.1 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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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 Neg10   branch if it is zero
        bchg  #31,d0  change sign bit
Neg10   rts           return
```

T$FSub — Single Precision Subtraction — T$FSub

ASSEMBLER CALL: TCALL T$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Sqrt Square Root Function T$Sqrt

ASSEMBLER CALL: TCALL T$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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$Math1,T$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 12

SEMAPHORES

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 is done with a semaphore.

A semaphore is a special type of system global variable. It is used as a tool to synchronize processes. There are two basic operations that may be performed on a semaphore: 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.

A semaphore can be thought of as a pigeon hole box for keys at a hotel. The numeric value of the semaphore 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 availabe.
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. Otherwise, (because of timeslicing) two processes could both check the semaphore and find it free and 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.

EVENTS AND 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.

Existing functions allow events to be created, deleted, linked, unlinked and examined. Several variations of the Signal and Wait operations are also provided.

The register conventions below apply to every event function. They are not repeated in the individual function specifications.

F$Event - Event system call.

INPUT: d1.w = function code

0	= Ev$Link	=	link to existing event
1	= Ev$UnLnk	=	unlink event
2	= Ev$Creat	=	create (and link to) new event
3	= Ev$Delet	=	delete existing event
4	= Ev$Wait	=	wait for event to occur
5	= Ev$WaitR	=	wait for relative event to occur
6	= Ev$Read	=	read event value
7	= Ev$Info	=	return event information
8	= Ev$Signl	=	signal event
9	= Ev$Pulse	=	pulse event
A	= Ev$Set	=	set event value
B	= Ev$SetR	=	set relative event value

Other registers depend on function code

OUTPUT: registers depend on function code

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

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

INPUT: (a0) = event name string pointer (max 11-chars)

OUTPUT: d0.1 = event ID number
(a0) = updated past event name

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

INPUT: d0.l = event ID number

OUTPUT: none

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

INPUT: d0.l = initial event variable value
d2.w = auto-increment for Ev$Wait
d3.w = auto-increment for Ev$Signl
(a0) = event name string pointer (max 11-chars)

OUTPUT: d1.l = event ID number
(a0) = updated past event name

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 made up of an always-incrementing event ID (the MS half of d0), and the index into the system's event table (the LS half). This number is used in subsequent F$Event calls to refer to the event created. The maximum number of active events is specified in the system configuration module ("init").

Ev$Delet - Delete existing event

INPUT: (a0) = event name string pointer (max 11-chars)

OUTPUT: (a0) = updated past event name

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

INPUT: d0.l = event ID number
d2.l = minimum activation value (signed)
d3.l = maximum activation value (signed)

OUTPUT: d1.l = actual event value

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

INPUT: d0.l = event ID number
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

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

INPUT: d0.l = event ID number

OUTPUT: d1.l = current event value

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

INPUT: d0.l = event index to begin search
(a0) = ptr to buffer for event information

OUTPUT: d0.l = event index found
(a0) = data returned in buffer

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 array subscript, 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 so a utility will be able to determine the status of all active events.

Ev$Signl - Signal an event occurrence

INPUT: d0.l = event ID number
d1.w = MS bit set to activate all processes in range

OUTPUT: none

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.

6v$Pulse - Signal an event occurrence

INPUT: d0.l = event ID number
d1.w = MS bit set to activate all processes in range
d2.l = event pulse value

OUTPUT: none

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 as a subroutine, and upon return the original event value is restored.

Ev$Set - Set event variable and signal an event occurrence

INPUT: d0.l = event ID number
d1.w = MS bit set to activate all processes in range
d2.l = new event value

OUTPUT: none

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

INPUT: d0.l = event ID number
d1.w = MS bit set to activate all processes in range
d2.l = (signed) increment for event variable

OUTPUT: none

POSSIBLE ERRORS: E$EvntID - the ID specified is not a valid active event

FUNCTION:

The Ev$SetR function is minor variation of Ev$Sign1. 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 13

SECTION 5 - SERVICE REQUEST DESCRIPTIONS

- User mode requests
- I/O requests
- System mode requests

INTRODUCTION TO THE SERVICE REQUEST DESCRIPTIONS

OS-9/68000 SERVICE REQUEST 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 mode function requests
2. I/O requests
3. System mode function requests

All system calls have a mnemonic name for easy reference. User and system functions names start with "F$", and I/O related requests begin with "I$". The mnemonic names are defined in the relocatable library file "sys.l" that should be linked with your programs.

The OS-9 I/O service requests 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 mode function requests 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 mode functions are included in this manual primarily for the benefit of those programmers who will be writing device drivers and other system-level applications.

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 accessable to users because they may occasionally be useful.

CROSS REFERENCE: See F$SchBit and F$DelBit.

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$CRC generates and checks the CRC (cyclic redundancy check) values of sections of memory. F$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$Chain Load and execute a new primary module F$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: 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$Chain is used when it is necessary to execute an entirely new program, but without the overhead of creating a new process. It is functionally similar to a Fork command followed by an EXIT. F$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.

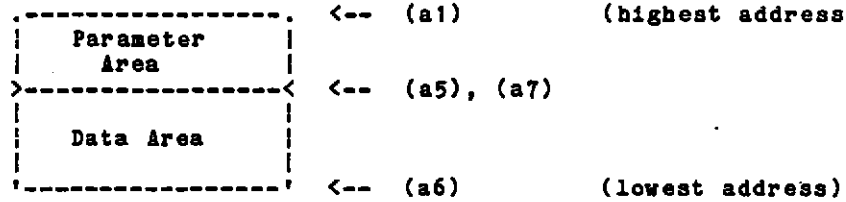
2. The system parses the name string of the new process' "primary module" (the program that will initially be executed). Next, the system module directory is searched to see if a module with the same name and type/language is already in memory. If so, the module is linked. If not, the name string is 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$Chain (Continued)

The diagram below shows how Chain sets up the data memory area and registers for the new module (these are identical to F$Fork).



Registers passed to child process:

- sr = 0000
- pc = module entry point
- d0.w = process ID
- d1.l = group/user number
- d2.w = priority
- d3.w = number of I/O paths inherited
- d4.l = undefined
- d5.l = parameter size
- d6.l = total initial memory allocation
- d7.l = undefined

- (a0) = undefined
- (a1) = top of memory pointer
- (a2) = undefined
- (a3) = primary (forked) module pointer
- (a4) = undefined
- (a5) = parameter pointer
- (a6) = static storage (data) base pointer
- (a7) = stack pointer (same as a5)

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$DatMod Create Data Module F$DatMod

ASSEMBLER CALL: 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$MNam

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 should be taken not to alter the data module's header and 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 256-byte 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 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$IProcID, E$ProAbt

FUNCTION: F$DExec controls the execution of a suspended child process that has been created by the F$DFork call. Control terminates when the specified number of instructions have been executed, a breakpoint is reached or an unexpected exception occurs.

Any OS-9 system calls made by the suspended program are executed at "full speed" and are considered one logical instruction.

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, tracing will no longer be permitted (carry bit will be set). An F$DExit call should be made.

CROSS REFERENCE: See F$DFork and F$DExit.

CAVEATS: Tracing is allowed through user 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: 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: 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. Whenever 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.

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 can 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 is alive, search wait queue of parent:
   - A. If parent can not be found, leave the process in limbo until parent notices the death.
   - B. If parent found, move parent to active queue, inform parent of death/status, remove child from sibling list and free process descriptor for the system.

F$Exit (continued)

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.

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.

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.

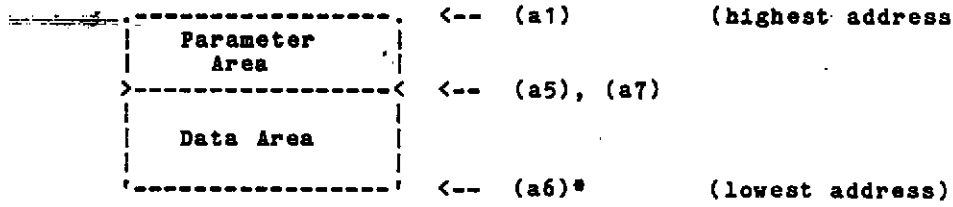
The new process' registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point.

When the shell processes a command line, it passes a string in the parameter area which is a copy of the parameter part (if any) of the command line. It also inserts an end-of-line character at the end of 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.

F$Fork (Continued)

The diagram below shows how Fork sets up the data memory area and registers for a newly-created process. / See F$Wait.



Registers passed to child process:

sr = 0000	(a0) = undefined
pc = module entry point	(a1) = top of memory pointer
d0.w = process ID	(a2) = undefined
d1.1 = group/user number	(a3) = primary (forked) module pointer
d2.w = priority	(a4) = undefined
d3.w = number of I/O paths inheirited	(a5) = parameter pointer
d4.1 = undefined	(a6) = static storage (data) base pointer*
d5.1 = parameter size	(a7) = stack pointer (same as a5)
d6.1 = total initial memory allocation	
d7.1 = undefined	

* (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$Wait call immediately after the fork, it will wait until the child dies before it resumes execution. Caution should be exercised when recursively calling a program that uses the F$Fork service request since another child may be created with each "incarnation" until the process table becomes full. A child process descriptor is returned only when the parent does an F$Wait call.

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.

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 processes.

CROSS REFERENCE: See F$GPrDBT.

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 system calls (such as I/O) should be avoided. In particular, signals should not be sent to the process. After the intercept routine has been completed, it may return to normal process execution by executing the F$RTE system call or the following code sequence:

```
movem.l  (a7)+,d0-d7/a0-a7     restore registers
RTR                            continue mainline execution
```

NOTE: The above code sequence is faster than the F$RTE system call.

CROSS REFERENCE: See F$RTE and F$Send.

CAVEATS: If another signal is received while a program is in its intercept routine, the intercept routine will be called recursively. 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 if 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 dates older than this, 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.

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 priveleged 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 256 byte boundary. Additional memory is allocated contiguously upward (towards higher addresses), or deallocated downward from 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 should contain 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

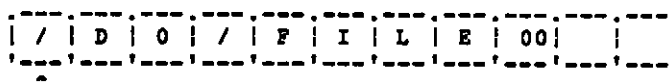
FUNCTION: F$PrsNam parses a string for a valid pathlist element, returning its size. Note that this does not parse an entire pathname, only one element in it. A valid pathlist element may contain the following characters:

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 that names may not be terminated by having the high order bit set (as on the 6809).

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.

BEFORE F$PrsNam CALL:



AFTER F$PrsNam CALL:

d0.b = "/"
d1.w = 2

CROSS REFERENCE: See F$CmpNam.

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.

This call is included for compatability with older systems (V1.1). In version 1.2 of OS-9, signal processing was changed to use the user's stack to keep the interrupted register stack image. It is far more efficient to use the following code to return from a signal processing routine:

```
movem.l  (a7)+,d0-d7,a0-a7     restore registers
rtr                            continue mainline execution
```

CAVEATS: When a signal is received, 70 bytes are used on the user stack. Signals may occur inside a signal processing routine, thus calling the routine recursively. Each time the routine is called, another 70 bytes are used. Consequently intercept routines must 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, or to all processes with the same group/user number. The signal code is a word value.

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.

NOTE: 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. The superuser (ID number 0) may send the same signal to all processes by the same technique. Notice that the super user is capable of aborting all processes by sending signal 0 to process 0.

F$Send (continued)

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 can be a handy but dangerous tool to get rid of unwanted background tasks.

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 10 on using the DEFS File.

CAVEATS: This call may only be used by the super user. Any system variable may be examined, but only a few may be altered. As of version 1.2, 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.

F$Sleep Put calling process to sleep F$Sleep

ASSEMBLER CALL: OS9 F$Sleep

INPUT: d0.l = Ticks/seconds (number of ticks 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$Sleep deactivates the calling process until the number of ticks requested have elapsed. Sleep(0) will sleep indefinately, Sleep(1) gives up a time slice but does not necessarily sleep for one tick. Due to the fact that it is not known when the F$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 indefinately 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.

CAVEATS: The system clock must be running to perform a timed sleep. The system clock is not required to perform an indefinate sleep or to give up a timeslice.

F$SPrior — Set Process Priority — F$SPrior

ASSEMBLER CALL: OS9 F$SPrior

INPUT: d0.w = Process ID number
d1.w = Desired process priority:
0 = lowest
65535 = highest

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 256 bytes for Level One). 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.

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. Memory must be returned block-by-block as it was granted. Any blocks not returned are automatically de-allocated by the system when a process terminates.

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 to 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 up 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 exits, 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

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 no 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-re-entrant.

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 process IDs 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 the super user.

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)	

F$Time (continued)

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 format:

```
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 (continued)

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 11 on Trap Handlers.

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 cannot 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.

ROMed modules should not be unlinked.

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 that the address of the module header.

CROSS REFERENCE: See F$UnLink.

F$Wait **Wait for child process to terminate** **F$Wait**

ASSEMBLER CALL: OS9 F$Wait

INPUT: none

OUTPUT: d0.w = Deceased child process' 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.

end of chapter 14

USER 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) = System's device table pointer

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 kernal attaches all devices at open, and detaches them at close.

I$Attach (continued)

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$Close terminates the I/O path specified by the path. The path number will no longer be valid for any OS-9 calls unless it becomes active again through an Open, Create, or Dup system call. When pathlists to devices that are non-sharable are closed, the devices become available to other requesting processes. If this is the last use of the path (i.e., it has not been inherited or dublicated by I$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 a multifile mass storage device. The new file name is entered in the directory structure. On non-multifile 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 in figure 23.

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.

I$Create (Continued)

Attribute bits	Mode bits
0 = read permit	0 = read
1 = write permit	1 = write
0 and 1 = update permit (read and write)	0 and 1 = update
2 = execute permit	2 = execute
3 = public read permit	5 = initial file size
4 = public write permit	6 = single user
3 and 4 = public update permit (public read and write)	7 = Directory
5 = public execute permit	if the bit is set, access is permitted.
6 = non-sharable file	

figure 23: access mode and attribute bit definitions

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$Dup returns a synonymous path number for the same file or device. The lowest available path number is used. I$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 SETSTT 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)

I$Getstt (continued)

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)

I$Getstt (Continued)

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)

I$Getstt (Continued)

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

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-multifile devices.

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:

Attribute bits	Mode bits
0 = read permit	0 = read
1 = write permit	1 = write
2 = execute permit	0 and 1 = update
3 = public read permit	2 = execute
4 = public write permit	6 = single user
5 = public execute permit	
6 = non-sharable file	If the bit is set, access is permitted.

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 (trailing spaces skipped).

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):

ACCESS BIT
0 = read
1 = write
0 and 1 = update (read and 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.

I$Open (continued)

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 during the open.

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$ReadLn Read a text line with editing I$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 (terminals) 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$Read Read data from a file or device I$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 and 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. An end-of-file occurs
B. An end-of-record occurs (SCF only)
C. An error condition occurs.

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 the next byte in the file to be read or written.

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
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
SS_DCmd	Direct command to hard disk controller (currently not implemented)

* These setstats exist in Microware disk drivers (if needed).

I$SetStt (Continued)

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

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.

I$SetStt (Continued)

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)	
INPUT	d0.w = Path number d1.w = #SS_RESET function code
OUTPUT	none
FUNCTION	This directs the disk head to track zero. It is used for formatting and for error recovery.

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.
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.

I$Setstt (Continued)

SS_FD: Write FD Sector (RBF)	
INPUT	d0.w = Path Number d0.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 (a0) = 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 indefinately 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.

I$SetStt (continued)

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 has data ready. SS_SSIG must be reset each time the signal is sent. The device 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.

I$SetStt (continued)

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_Sig, 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

I$SetStt (continued)

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 goes true.

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$WriteLn Write a line of text with editing **I$WriteLn**

ASSEMBLER CALL: OS9 I$WriteLn

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$WritLn and Chapter 7 on SCF Drivers (line editing).

CAVEATS: On RBF devices, any record that was locked is released.

end of chapter 15

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 scheduled 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 it's 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 MODE SERVICE REQUEST

F$AllPD Allocate process/path descriptor F$AllPD

ASSEMBLER CALL: OS9 F$AllPD

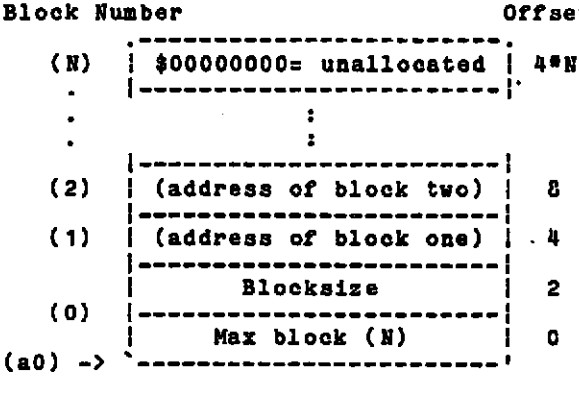
INPUT: (a0)=process/path table ptr

OUTPUT: d0.w=process/path number
(a1)=ptr 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:



CRPSS REFERENCE: See F$FindPD and F$RetPD.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE 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$ALLProc 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 Level One systems, this is a direct call to F$AllPD.

CROSS REFERENCE: See F$AllPD

NOTE: THIS IS A PRIVILEGED SYSTEM MODE 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 ptr

OUTPUT: (a1)=ptr 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 MODE 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$Find64, F$Send*, and F$Sleep.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE 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 ptr
(a3) = port address
(a6) = system global data ptr (D_'s)
(a7) = system stack (in active proc's descriptor)
```

ERROR OUTPUT: Carry bit set if the device did not cause the interrupt. May destroy any registers except (a7) and (usp).

CROSS REFERENCE: See Chapter 7 and Chapter 8 for more information on RBF and SCF device drivers.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$Move — Move Data (low bound first) — F$Move

ASSEMBLER CALL: OS9 F$Move

INPUT:
d0.w = Source task number (not required on Level 1)
d1.w = Destination task number (not required on Level 1)
d2.l = 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 moves data bytes from one address space to another, usually from system to user or vice versa. On Level One systems, the task numbers are not used.

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 MODE SERVICE REQUEST

F$NPrc Start next process F$NPrc

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 MODE 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 ptr

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 MODE SERVICE REQUEST

F$SLink System Link F$SLink

ASSEMBLER CALL: OS9 F$SLink

INPUT: d0.w = Desired module type/language (0 = any)
(a0) = Module name string pointer

OUTPUT: d0.w = Actual module type/language
d1.w = Module attributes/revision
(a0) = Updated beyond name string
(a1) = Module entry point
(a2) = Module pointer

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

POSSIBLE ERRORS: E$ModBsy, E$MemFul

FUNCTION: F$SLink links a module whose name is outside the current (system) process' address space into the address space that contains its name. On Level One systems this is synonymous with F$Link.

CROSS REFERENCE: See F$FModul and F$Link..

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST.

F$SSvc **Service request table initialization** **F$SSvc**

ASSEMBLER CALL: OS9 F$SSvc

INPUT: (a1) = pointer to service request initialization table

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.

An example initialization table might look like this:

```
SvcTbl
       dc.w F$Service          OS-9 service request code
       dc.w Routine-*-4        offset of routine to process request
              :
       dc.w F$Service+SysTrap  redefine system level request
       dc.w SysRoutn-*-4       offset of routine to handle sys 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 (continued)

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 MODE 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 MODE SERVICE REQUEST

end of chapter 16

USER NOTES

Chapter 14 - User Mode Function Requests

end of appendix a

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.

RBF Descriptor Example

```
Microware OS-9/68000 Resident Macro Assembler V1.5  85/07/25 15:53 Page 1
D0 Device Descriptor - Device Descriptor for Floppy disk controller
00001                           nam     D0              Device Descriptor
00002                           use     defsfile
00008
00009
00003                           use     ../io/rbfdesc.a
00001
00002                           ttl     Device      Descriptor for Floppy disk
00003
00004 * Copyright 1984, 1985 by Microware Systems Corporation.
00005 * Reproduced Under License.
00006
00007 * This source code is the proprietary confidential property of
00008 * Microware Systems Corporation, and is provided to licensee
00009 * solely for documentation and educational purposes. Reproduction,
00010 * publication, or distribution in any form to any party other
00011 * than the licensee is prohibited.
00012
00013 ********************************
00014 * Edition History
00015 *  #   date     comments                                                by
00016 * -- -------- ------------------------------------------------------- ---
00017 * 00 12-05-83 Converted to 68000 from 6809 edition 1.                  res
00018 * 00 04-06-84 Added FlpyName macro usage                               WGP
00019 * 00 04-06-84 Added defs for some variable opts                        WGP
00020 * 01 10-12-84 Added IRQ Level & resesrved bytes.                       rfd
00021 * 02 11-07-85 Split into seperate files                                res
00022 * 03 06-27-85 Added mode byte.rfd
00023 * 04 06-25-85 Added sector size, format control, retry control.res
00024  00000004 Edition    equ     4           current edition number
00025
00026  00000000 Single     equ     0
00027  00000001 Double     equ     1
00028  00000000 Five       equ     0
00029  00000001 Eight      equ     1
00030  00000080 Hard       equ     $80
00031  00000001 ON         equ     1
00032  00000000 OFF        equ     0
00033
00034  00000001 d877       equ     1           single density 8"
00035  00000004 dd877      equ     4           double density 8"
```

```
00036  00000002 d540        equ      2          single density 5" 40 trk
00037  00000005 dd540       equ      5          double density 5" 40 trk
00038  00000003 d580        equ      3          single density 5" 80 trk
00039  00000006 dd580       equ      6          double density 5" 80 trk
00040
00041  00000001 Density     set      BitDns+(TrkDns<<1)
00042  00000001 DiskType    set      DiskKind+(DnsTrk0<<5)
00043
00044  00000f00 TypeLang    set      (Devic<<8)+0
00045  00008000 Attr_Rev    set      (ReEnt<<8)+0
00046                       psect    RBFDesc,TypeLang,Attr_Rev,Edition,0,0
00047
00048 0000 001f             dc.l     Port       port address
00049 0004   1b             dc.b     Vector     auto-vector trap #
00050 0005   03             dc.b     IRQLevel   IRQ interrupt level
00051 0006   02             dc.b     Priority   IRQ polling priority
00052 0007   a7             dc.b     Mode       device mode capabilities
00053 0008 0031             dc.w     FileMgr    file manager name offset
00054 000a 0035             dc.w     DevDrv     device driver name offset
00055 000c 0000             dc.w     DevCon     (reserved)
00056 000e 0000             dc.w     0,0,0,0    reserved
00057 0016 0019             dc.w     OptLen
00058
00059 * Default Parameters
00060                       OptTbl
00061 0018=   00            dc.b     DT_RBF     device type
00062 0019    00            dc.b     DrvNum     drive number
00063 001a    03            dc.b     StepRate   step rate
00064 001b    01            dc.b     DiskType   type of disk 8"/5"/Hard
00065 001c    01            dc.b     Density    Bit Dens and track density
00066 001e 004d             dc.w     Cylnders-TrkOffs    # of cylinders
00067 0020    02            dc.b     Heads      # Sides(Floppy)Heads(Hard)
00068 0021    00            dc.b     NoVerify   OFF =verify  ON =no verify
00069 0022 001c             dc.w     SectTrk    default sectors/track
00070 0024 0010             dc.w     SectTrk0   default sectors/track0
00071 0026 0008             dc.w     SegAlloc   segment allocation size
00072 0028    03            dc.b     Intrleav   sector interleave factor
00073 0029    00            dc.b     DMAMode    DMA mode (none)
00074 002a    00            dc.b     TrkOffs    track base offset
00075 002b    00            dc.b     SectOffs   sector base offset
00076 002c 0100             dc.w     SectSize   # of bytes/sector
00077 002e 0000             dc.w     Control    format control byte
00078 0030    07            dc.b     Trys       # of retrys 0 =no retrys
00079  00000019 OptLen       equ      *-OptTbl
00080
00081 0031 5242 FileMgr     dc.b     "RBF",0    Random block file manager
00082                       RBFDesc  macro
00083
00084                       Port     equ      \1         Port address
00085                       Vector   equ      \2         autovector number
00086                       IRQLevel equ      \3         hardware interrupt level
```

```
00087           Priority    equ     \4          polling priority
00088           DevDrv      dc.b    "\5",0      driver module name
00089                       ifgt    \#-5        stnd dev setup requested?
00090
00091
00092                       ifeq    \6-d877     8", 77 track drive
00093           DiskKind    set     Eight
00094           Cylnders    set     77
00095           TrkDns      set     Single
00096           SectTrk     set     16
00097           SectTrk0    set     16
00098           DevCon      set     0
00099                       endc
00100
00101                       ifeq    \6-dd877    8", 77 track,double density
00102           DiskKind    set     Eight
00103           Cylnders    set     77
00104           BitDns      set     Double
00105           TrkDns      set     Single
00106           SectTrk     set     28
00107           SectTrk0    set     16
00108           DevCon      set     0
00109                       endc
00110
00111                       ifeq    \6-d540     5", 40 track drive
00112           DiskKind    set     Five
00113           Cylnders    set     40
00114           BitDns      set     Single
00115           TrkDns      set     Single
00116           SectTrk     set     10
00117           SectTrk0    set     10
00118           DevCon      set     0
00119                       endc
00120
00121                       ifeq    \6-dd540    5", 40 track,double density
drive
00122           DiskKind    set     Five
00123           Cylnders    set     40
00124           BitDns      set     Double
00125           TrkDns      set     Single
00126           SectTrk     set     16
00127           SectTrk0    set     10
00128           DevCon      set     0
00129                       endc
00130
00131                       ifeq    \6-d580     5", 80 track,double density
drive
00132           DiskKind    set     Five
00133           Cylnders    set     80
00134           BitDns      set     Single
00135           TrkDns      set     Double
```

```
00136            SectTrk    set       10
00137            SectTrk0   set       10
00138            DevCon     set       0
00139                       endc
00140
00141                       ifeq      \6-dd580   5", 80 track drive
00142            DiskKind   set       Five
00143            Cylnders   set       80
00144            BitDns     set       Double
00145            TrkDns     set       Double
00146            SectTrk    set       16
00147            SectTrk0   set       10
00148            DevCon     set       0
00149                       endc
00150
00151                       endc
00152                       endm
00153
00154 ****************
00155 * Descriptor Defaults
00156  000000a7 Mode       set       Dir_+ISize_+Exec_+Updat_
00157  00000000 BitDns     set       Single
00158  00000002 Heads      set       2
00159  00000002 StepRate   set       2
00160  00000003 Intrleav   set       3
00161  00000000 NoVerify   set       OFF
00162  00000000 DnsTrk0    set       Single
00163  00000000 DMAMode    set       0          non dma device
00164  00000008 SegAlloc   set       8          min segment alloc size
00165  00000000 TrkOffs    set       0
00166  00000000 SectOffs   set       0
00167  00000100 SectSize   set       256        default sect size 256 byte
00168  00000000 FmtEnabl   set       0          enable formatting
00169  00000001 FmtDsabl   set       1          disable formatting
00170  00000000 Control    set       FmtEnabl   enable formatting
00171
00172  00000007 Trys       set       7          number of Trys
00173
00004  00000000 DrvNum     set       0
00005                      DiskD0
00006  0000003c            ends
Errors: 00000
Memory used: 21k
Elapsed time: 15 second(s)
```

SCF Descriptor Example

```
Microware OS-9/68000 Resident Macro Assembler V1.5  85/07/25 15:54 Page 1
Term - 68000 Term device descriptor module
00001                           nam       Term
00002                           ttl       68000        device descriptor module
00003                           use       defsfile
00008
00009
00004
00005 * The default characteristics in "scfdesc.a" can be overridden
00006 * by equating the desired values here.  For example:
00007
00008                           use       ../io/scfdesc.a
00001 ********************************
00002 * Edition History
00003 *  #    date     comments                                                  by
00004 * -- -------- ------------------------------------------------------ ---
00005 * 00 09-28-83 Converted to 68000 from 6809 source                       rfd
00006 * 00 04-06-84 Added use of TrmDrNam macro for driver name               WGP
00007 * 01 10-12-84 Added IRQ Level & resesrved bytes.                        rfd
00008 * 02 10-24-84 Changed to "use" file format.                             rfd
00009 * 03 11-05-84 Inserted macro for descriptor generation.                 rfd
00010 * 04 06-27-85 Added mode byte.rfd
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 0060            dc.l      Port              port address
00019 0004   1d            dc.b      Vector            auto-vector trap #
00020 0005   05            dc.b      IRQLevel          IRQ interrupt level
00021 0006   fa            dc.b      Priority          IRQ polling priority
00022 0007   23            dc.b      Mode              Dev mode capabilities
00023 0008 0034            dc.w      FileMgr           file manager name offset
00024 000a 0038            dc.w      DevDrv            dev driver name offset
00025 000c 0000            dc.w      0                 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=   00            dc.b      bsb       backspace=BS,SP,BS           ON
00037 001b=   00            dc.b      linedel   line del/bsp line           OFF
```

```
00038 001c=   00              dc.b     autoecho  full duplex              ON
00039 001d=   00              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=   00              dc.b     pagsize   lines per page           24
00043 0021=   00              dc.b     C$Bsp     backspace char           ^H
00044 0022=   00              dc.b     C$Del     delete line char         ^X
00045 0023=   00              dc.b     C$CR      end of record char <return>
00046 0024=   00              dc.b     C$EOF     end of file char         ESC
00047 0025=   00              dc.b     C$Rprt    reprint line char        ^D
00048 0026=   00              dc.b     C$Rpet    dup last line char       ^A
00049 0027=   00              dc.b     C$Paus    pause char               ^W
00050 0028=   00              dc.b     C$Intr    Keyboard Interrupt char  ^C
00051 0029=   00              dc.b     C$Quit    Keyboard Quit char       ^E
00052 002a=   00              dc.b     C$Bsp     backspace echo char      ^H
00053 002b=   00              dc.b     C$Bell    line overflow char       ^G
00054 002c    00              dc.b     Parity    stop bits and parity     none
00055 002d    0e              dc.b     BaudRate  bits/char & baud rate    none
00056 002e=0000               dc.w     EchoNam   offset of echo device    none
00057 0030=   00              dc.b     C$XOn     Transmit Enable char     ^Q
00058 0031=   00              dc.b     C$XOff    Transmit Disable char    ^S
00059 0032=   00              dc.b     C$Tab     tab character            ^I
00060 0033=   00              dc.b     tabsize   tab column size           4
00061  0000001c OptSiz        equ      *-Options
00062
00063 0034 5363 FileMgr       dc.b     "Scf",0       file manager
00064
00065 * Macro to generate main features of device descriptor
00066            SCFDesc      macro
00067                         ifne     \#-7          must have exactly 7 args
00068                         fail     SCFDesc:      must spec all arguments
00069                         endc
00070
00071            Port         equ      \1            Port address
00072            Vector       equ      \2            autovector number
00073            IRQLevel     equ      \3            hardware interrupt level
00074            Priority     equ      \4            polling priority
00075            Parity       equ      \5            parity, stop bits
00076            BaudRate     equ      \6            baud rate
00077            DevDrv       dc.b     "\7",0        driver module name
00078            EchoNam      equ      bname         echo device descriptor
00079                         endm
00080
00081  00000023 Mode          set      ISize_+Updat_ def dev mode capability
00082
00009                         TERM
00010  00000040               ends
00011
Errors: 00000
Memory used: 21k
Elapsed time: 11 second(s)
```

INIT Module Example

```
Microware OS-9/68000 Resident Macro Assembler V1.5  85/07/25 15:54 Page 1
Init: OS-9 Configuration Module -
00001                           nam         Init:   OS-9 Configuration Module
00002 ********************************
00003 * Editon History
00004 *  # Date      Comments                                                 by
00005 * -- -------- ------------------------------------------------------- ---
00006 * 00 12-01-83 Initial test version developed.                          rfd
00007 * 01 01-20-84 Added code to process startup file.                      rfd
00008 * 02 11-01-84 Changed names and added new information.                 rfd
00009 * 03 12-20-84 Changed slice value from 1 to 2.                         rfd
00010 * 04 06-21-85 Minor changes for V1.2.rfd
00011  00000004 Edition    equ       4                 current edition number
00012
00013  00000c00 Typ_Lang   set       (Systm<<8)+0
00014  00008000 Attr_Rev   set       (ReEnt<<8)+0
00015                      psect     Init,Typ_Lang,Attr_Rev,Edition,0,0
00016
00017 * Config constants (default; changable in "systype" file)
00018  000109a0 CPUTyp     set       68000     cpu type (68008/68000/68010)
00019  00000001 Level      set       1         OS-9 Level One
00020  00000001 Vers       set       1         Version 1.2
00021  00000002 Revis      set       2
00022  00000000 Edit       set       0         Edition
00023  00000000 Site       set       0         Installation Site code
00024  00000080 MDirSz     set       128       module dir size (unused)
00025  00000020 PollSz     set       32        IRQ poll table size (fixed)
00026  00000020 DevCnt     set       32        device table size (fixed)
00027  00000040 Procs      set       64        init process table size
00028  00000040 Paths      set       64        init path table size
00029  00000002 Slice      set       2         ticks per time slice
00030  00000080 SysPri     set       128       init system priority
00031  00000000 MinPty     set       0         init system min exec prior
00032  00000000 MaxAge     set       0         init sys max age limit
00033  00000000 MaxMem     set       0         Top of RAM (unused)
00034  00000000 Events     set       0         init event table size
00035
00036                      use       defsfile (above may be overridden)
00008
00009
00037
00038 * Configuration module body
00039 0000 0000            dc.l      MaxMem    (currently unused)
00040 0004 0020            dc.w      PollSz    IRQ polling table size
00041 0006 0020            dc.w      DevCnt    device table size
00042 0008 0040            dc.w      Procs     process table size
00043 000a 0040            dc.w      Paths     path table size
00044 000c 0000            dc.w      0         I/O manager mod name offset
00045 000e 008c            dc.w      SysStart  executable mod name offset
```

```
00046 0010 0092              dc.w     SysDev     sys default dev name offset
00047 0012 0096              dc.w     ConsolNm   stnd I/O path name offset
00048 0014 006a              dc.w     Extens     Customation mod name offset
00049 0016 009c              dc.w     ClockNm    clock module name offset
00050 0018 0002              dc.w     Slice      num of ticks per time slice
00051 001a 0070              dc.w     UsrAct     account package name offset
00052 001c 0000              dc.l     Site       installation site code
00053 0020 0076              dc.w     MainFram   installation name offset
00054 0022 0001              dc.l     CPUTyp     68000 type processor in use
00055 0026 0101              dc.b     Level,Vers,Revis,Edit OS-9 Level
00056 002a 0056              dc.w     OS9Rev     OS-9 revision string offset
00057 002c 0080              dc.w     SysPri     initial system priority
00058 002e 0000              dc.w     MinPty     init sys min exec priority
00059 0030 0000              dc.w     MaxAge     max sys natural age limit
00060 0032 0000              dc.l     MDirSz     mod directory size (unused)
00061 0036 0000              dc.w     Events     init event table size
00062 0038 0000              dc.w     0,0,0,0,0,0,0  reserved
00063 0046 0000              dc.w     0,0,0,0,0,0,0,0 reserved
00064
00065 * Configuration name strings
00066 0056 4f53 OS9Rev       dc.b     "OS-9 Level One V1.2",0
00067 006a 4f53 Extens       dc.b     "OS9P2",0    Customization module name
00068 0070 5541 UsrAct       dc.b     "UAcct",0    user account module name
00069
00070 * The remaining names are defined in the "systype" macro below
00071 -                      CONFIG
00072  000000a4              ends
00073
Errors: 00000
Memory used: 21k
Elapsed time: 9 second(s)
```

SYSGO Module Example

```
Microware  OS-9/68000 Cross Macro Assembler V1.3  84/11/11  16:59  Page 1
sysgo.a
Sysgo - OS-9/68000 Initial (startup) module
                    nam     Sysgo
                    ttl     OS-9/68000       Initial (startup) module
**********************************

 00000001 Edition   equ     1                current edition number

 00000101 Typ_Lang  set     (Prgrm<<8)+Objct
 00000000 Attr_Rev  set     0                (non-reentrant)
                    psect   test,Typ_Lang,Attr_Rev,Edition,0,Entry

                    use      defsfile
* generic defsfile, no hardware dependent conditions allowed



                    vsect
 00000000           ds.b    255              stack space
 00000000           ends

 00000080 Priority  equ     128              initial priority

0000=4e40 Intercpt  os9     F$RTE            return from intercept

0004 41fa Entry     lea     Intercpt(pc),a0
0008=4e40           os9     F$Icpt
000c 41fa           lea     CmdStr(pc),a0    default execution dir ptr
0010=7000           moveq   #Exec_,d0         execution mode
0012=4e40           os9     I$ChgDir          change exec dir
0016 640c           bcc.s   Entry10           continue if no error
0018 7001           moveq   #1,d0             std output path
001a 7218           moveq   #ChdErrSz,d1      size
001c 41fa           lea     ChdErrMs(pc),a0   "I can't find CMDS"
0020=4e40           os9     I$WritLn          output error message

* Process startup file
0024 7000 Entry10   moveq   #0,d0             any type module
0026 7200           moveq   #0,d1             default memory size
0028 7408           moveq   #StartSiz,d2      size of startup command
002a 7603           moveq   #3,d3             copy std I/O paths
002c 383c           move.w  #Priority,d4      medium priority
0030 41fa           lea     ShellStr(pcr),a0  shell name
0034 43fa           lea     StartStr(pcr),a1  startup pathlist
0038=4e40           os9     F$Fork            fork shell
003c 6410           bcc.s   Entry20           continue if no error
```

```
Microware OS-9/68000 Cross Macro Assembler V1.3  84/11/11  16:59   Page 2
Sysgo - OS-9/68000 Initial (startup) module
003e 7001            moveq   #1,d0              std output path
0040 721a            moveq   #FrkErrSz,d1       size
0042 41fa            lea     FrkErrMs(pc),a0    "can't fork Shell"
0046=4e40            os9     I$WritLn           output error message
004a=4e40            os9     F$SysDbg           crash

004e=4e40 Entry20    os9     F$Wait             wait, ignore any error

0052 7000 Loop       moveq   #0,d0              any type module
0054 7200            moveq   #0,d1              default memory size
0056 7401            moveq   #1,d2              one parameter byte (CR)
0058 7603            moveq   #3,d3              copy std I/O paths
005a 383c            move.w  #Priority,d4       medium priority
005e 41fa            lea     ShellStr(pcr),a0   shell name
0062 43fa            lea     CRChar(pcr),a1     null parameter string
0066=4e40            os9     F$Fork             fork shell
006a 650a            bcs.s   ForkErr            abort if error
006c=4e40            os9     F$Wait             wait for it to die
0070 6504            bcs.s   ForkErr
0072 4a41            tst.w   d1                 zero status?
0074 67dc            beq.s   Loop               loop if so
0076=4e40 ForkErr    os9     F$PErr             print error message
007a 60d6            bra.s   Loop

007c 5379 FrkErrMs   dc.b    "Sysgo can't fork to "
0090=5368 ShellStr   dc.b    "Shell",C$CR
 0000001a FrkErrSz   equ     *-FrkErrMs

0096 5379 ChdErrMs   dc.b    "Sysgo can't chx to "
00a9=434d CmdStr     dc.b    "CMDS",C$CR
 00000018 ChdErrSz   equ     *-ChdErrMs

00ae 7374 StartStr   dc.b    "startup"
00b5=  00 CRChar     dc.b    C$CR
 00000008 StartSiz   equ     *-StartStr

 000000b6            ends

Errors: 00000
Memory used: 10k
Elapsed time: 15 second(s)
```

end of appendix b

OS-9 ERROR CODES

ERROR NUMBER		DESCRIPTION
000:200	E$PthFul	PATH TABLE FULL - This error is generally returned when a user program has tried to open more than 32 I/O paths simultaneously. When the system path table becomes full, the kernel automatically expands it. However, this error could be returned if there is not enough (contiguous) memory to expand it.
000:201	E$BPNum	ILLEGAL PATH NUMBER - This error is returned when the path number was too large or for a nonexistant 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.

ERROR NUMBER		DESCRIPTION
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.
000:206	E$DirFul	MODULE DIRECTORY FULL - This error is returned when the specified module can not be added to the system, because the module directory is full. To load or create another module, one must first be unlinked. While OS-9 expands the module directory when it becomes full, this error may be returned because there is not enough memory or the memory is too fragmented to use.
000:207	E$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 alloacated the maximum number of blocks permitted by the system. This could occur when trying to load a module.

ERROR NUMBER		DESCRIPTION
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$BPAddr	BOUNDARY ERROR - This error is returned when a memory allocation or deallocation request is not on a page boundary or an attempt is made to deallocate memory not previously assigned.
000:211	E$EOF	END OF FILE - This error is returned when an end of file condition is encountered on a read operation.
000:212	E$VctBsy	VECTOR BUSY - This error is returned when a device is trying to use an IRQ vector that is currently being used by another device.
000:213	E$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.

ERROR NUMBER		DESCRIPTION
000:214	E$FNA	FILE NOT ACCESSABLE - This error is returned when trying to open a file or device without the correct access permissions. Check the file's attributes and the owner ID.
000:215	E$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 error 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. This can also occur on a disk that has little free memory left or one whose free memory is too scattered. The simplest way to solve this problem is to copy the file (or disk), which should move it into more contiguous areas.
000:218	E$CEF	FILE ALREADY EXISTS - This error occurs when trying to create or duplicate a file using a name that already appears in the current directory.

ERROR NUMBER		DESCRIPTION
000:219	E$IBA	ILLEGAL BLOCK ADDRESS - This error is returned when a search for an illegal block address has occurred. An invalid pointer or block size has been passed or the device's file structure is damaged.
000:220	E$Hangup	TELEPHONE (MODEM) DATA CARRIER LOST
000:221	E$MNF	MODULE NOT FOUND - This error is returned when a request is made to link to a module that is not found in the module directory.
000:222	E$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 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$IProID	ILLEGAL PROCESS NUMBER - This error is returned when a system call is passed a process ID to a non-existant process or a process that the user may not access.

ERROR NUMBER		DESCRIPTION
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$Wait request is made and the process can not wait for its child process' I/O, because it has no child process.
000:227	E$ITrap	ILLEGAL TRAP CODE - This error is returned when an unavailable (already in use) or invalid trap code is used in a TLINK call.
000:228	E$PrcAbt	PROCESS ABORTED - This error is returned when a process is aborted by the signal code: 000:0.
000:229	E$PrcFul	PROCESS TABLE FULL - This error is returned when the system process table is full (too many processes currently running). While 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 the ridiculous parameters are passed to fork call.

ERROR NUMBER		DESCRIPTION
000:231	E$KwnMod	KNOWN MODULE - This error is returned when a call is made to install a module that is already in memory.
000:232	E$BMRC	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$USigP	SIGNAL ERROR - This error is returned by F$Send when the receiving process has a previous unprocessed signal pending. The sending process should try again later.
000:234	E$NEMod	NON-EXISTENT MODULE - This error is returned when a process tries to search for an unlocatable module. This might occur when using F$Chain or F$Link.
000:235	E$BNam	BAD NAME - This error occurs when there is a syntax error in the specified name.
000:236	E$BMHP	BAD PARITY - This error is returned when the specified module has bad module header parity.

ERROR NUMBER		DESCRIPTION
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. This error will not occur on OS-9 Level One systems.

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.

ERROR NUMBER		DESCRIPTION
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$BType	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
000:250	E$DevBsy	DEVICE BUSY - non-sharable device is in use.
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.

ERROR NUMBER		DESCRIPTION
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.

UNINITIALIZED TRAP ERRORS

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	TRACE ERROR - uninitialized trace exception occurred.

ERROR NUMBER	DESCRIPTION
000:110 E$1010	1010 TRAP - Line 1010 emulator exception occurred.
000:111 E$1111	1111 TRAP - Line 1111 emulator exception occurred.
000:112 - 000:123 E$Resrvd	reserved: an invalid TRAP (Ø12 - 23) occurred.
000:124 - 000:138 E$Trap	uninitialized user TRAP 1-15 executed.

OTHER ERRORS

ERROR NUMBER		DESCRIPTION
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.
000:139	E$Permit	NO PERMISSION - you must be super user to perform the requested function.

ERROR NUMBER		DESCRIPTION
000:140	E$Differ	DIFFERENT ARGUMENTS - the arguments to F$ChkNam do not match.
000:141	E$StkOvf	STACK OVERFLOW - F$ChkNam can cause this error if the pattern string is too complex.
000:142	E$EvntID	ILLEGAL EVENT ID - This error is returned when an invalid or illegal event ID number is specified.
000:143	E$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.
000:145	E$EvBusy	EVENT BUSY - This error is returned when an attempt to delete an event is made and its link count is non-zero. This can also occur if an attempt to create an already existant named event is made.
000:146	E$EvParm	IMPOSSIBLE EVENT PARAMETER - This error is returned when impossible parameters are passed to F$Event.

end of appendix c

USER NOTES

INDEX

For an alphabetical listing of OS-9 System Calls, see Appendix C.

INDEX

INDEX

MICROWARE SYSTEMS CORPORATION

DES MOINES

Microware®

[redacted]

OS-9/68000

ADVANCED SYSTEM

SOFTWARE

MICROWARE
RELEASE NOTES
OS-9/68000
VERSION 1.2

Publication date: July, 1985
Publication Editor: Walden Miller

Microware Systems Corporation
1866 NW 114th Street
Des Moines, Iowa 50322
Tel: (515)224-1929

OVERVIEW

This release of the 68000 version of OS-9 contains many updates from previous versions of the software. This document attempts to provide a thorough presentation of the changes that have occurred. This section will provide a brief summary of the major changes that have occurred. Separate sections will be devoted to specifying the detailed changes in major portions of the software.

The major portions of the system where changes have occurred are as follows:

1. The OS-9/68000 kernel
2. The OS-9/68000 drivers
3. The OS-9/68000 math libraries
4. The OS-9/68000 utilities
5. The OS-9/68000 Scred Screen Editor
6. The BASIC09/68000 programming language
7. The OS-9/68000 C compiler
8. The OS-9/68000 documentation

The remainder of this section will present the major changes that have occurred in each of these areas.

The OS-9/68000 Kernel

Several major changes have occurred in the OS-9/68000 kernel for the 1.2 release. The most notable are the addition of event (semaphore) routines and named pipes in the kernel. New sections have been added to the OS-9/68000 Technical Manual to describe these features. The naming conventions for filenames has been changed, with several characters being added to the list of allowable characters in filenames.

Numerous bug fixes have been made to provide a more stable programming environment.

MICROWARE RELEASE NOTES
OVERVIEW
OS-9/68000 VERSION 1.2

The OS-9/68000 Drivers

Several major improvements have been made in the OS-9 drivers for this release. Specifically, the TMODE and XMODE utilities have been upgraded to allow for the specification of baud rates with corresponding changes in the drivers and the serial port drivers have been upgraded to allow for modem control.

The OS-9/68000 Math Libraries

The version 1.1 math package consisted of two different trap handling routines to provide floating point math support. The new version has combined those trap handlers into a single module that contains all of the math routines. Numerous bug fixes have been made in the math routines and several routines have been rewritten to provide better performance. Existing programs should execute correctly with the new math module and will not require recompilation.

The OS-9/68000 Utilities

Numerous bug fixes have been made in the OS-9 utilities. Some enhancements have also been made. The most substantial change is a change in SHELL to provide a UNIX-like environment variable facility that can be accessed by C programs. The DSAVE utility has been upgraded to allow for the specification of dates and times for baselines of when to backup files.

OS9/68000 Scred Screen Editor

SCRED has been enhanced to improve performance, in particular in the area of I/O. Most changes have been made to be upward compatible. One exception to this is the TERMSET file used for terminal configurations by SCRED. The format for this file has changed and the user is referred to the "OS-9/68000 SCRED USER'S MANUAL" for further details.

The BASIC09/68000 Programming Language

Numerous bug fixes have been made in the BASIC09 language. When combined with the fixes in the math libraries, it is a much more reliable product. None of the changes is substantial enough in nature to warrant further discussion here.

The OS-9/68000 C Compiler

The major addition to the C compiler is a new data type "remote" that allows for the specification of very large arrays.

The OS-9/68000 Documentation

All of the previous documentation has been rewritten for this new release to reflect the changes in the operating system and the OS-9 languages. The format for each of the documents has been standardized and indexes have been included for the first time. The text has been rewritten to be more easily understood by new users of OS-9. Several sections of various manuals have been greatly expanded. This includes an expanded set of documentation on the math libraries in the Technical Manual, a greatly expanded tutorial section in the BASIC09 Manual and a rewritten tutorial section in the User's Manual.

KERNEL CHANGES

SYSTEM CALL CHANGES

F$Aproc: This service request has been rewritten using a more efficient algorithm. Process priorities and ages behave logically the same as they used to, but have been implemented using a more efficient algorithm.

F$CRC: A process normally does not give up its time slice while it is in the middle of a system call. Some long system calls take so long to process that this gives poor real-time response to other processes. The worst case is probably F$CRC, which is called every time a module is loaded to verify the module CRC. This call has been modified to periodically give up its time slice.

F$DExec: This call had a few minor problems: while single stepping, the debugger would not stop at the first instruction of a user's intercept routine. The same problem prevented the debugger from stopping at the first instruction of a user routine to handle "hardware" errors (such as zero divide). These problems have been corrected.

F$Event: This new system call provides the ability to create named system semaphores (events). See The OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL (chapter 13) for details.

F$Load: A problem was found and fixed that could crash the system when a file containing several modules was loaded. If the modules were already in memory, the system would get stuck in a dead loop in system state.

The "mode" parameter is now allowed. This allows modules to be loaded from the current data directory instead of the execution directory.

To improve system security, F$Load will no longer permit a user to load modules owned by a super user, unless the file they reside in is also owned by the super user. This change, along with changes to F$SUser make it more difficult for a normal user to become "super".

F$PrsNam: A change was made to extend the syntax of system names. This applies to both module names and path names. The new syntax recognizes the following characters as valid:

"A-Z" "a-z" "0-9" "." "_" "$"

A name must include at least one letter or digit to be valid. Names may begin with non-alphameric characters now.

F$Send: This system call had a problem that made it unusable in "broadcast" mode (ie. the destination process ID was zero). This has been fixed. If a user enters "kill 0" at an OS-9 shell prompt, all processes owned by that user will be terminated, except the shell itself. See the system call description in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" for more information.

F$SetSys: This system call had problems when changing either D_MinPty or D_MaxAge. When either of these variables is now changed, the active queue is re-ordered if necessary.

F$Sleep: In version 1.1, if a timed sleep was made with the sleep interval specified in seconds, the system converted the interval to ticks incorrectly. This was only a problem with very long sleep intervals. This has been corrected.

The coordination problem between F$Sleep and F$Send has been corrected. It was possible for a process doing a timed sleep to have its sleep interval decremented to $FFFFFFFF, and sleep for an extremely long time. This made timed sleep very unstable.

F$SPrior: In version 1.1, any process could change the priority of any other process. Now you must have the same user number to change a process' priority. A super user may still change the priority of any process.

F$SPrior now resorts the active process queue if you change the priority of an active process.

F$STrap: This call has been extended to allow user programs to catch bus and address trap errors.

It now has a method to remove or replace trap handling routines that are no longer needed.

F$SUser: To help increase security, this system call has been given restrictions on who may change their user number. See the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" (chapter 14) for more details.

F$Time: In version 1.1, when asked to return the current tick, it returned an incorrect (constant) number. This has been corrected.

Misc: In version 1.1, F$FindPD and F$AllPD were not using the PD block table pointer passed in (a0). These calls were also defined in the defs files as F$F64 and F$A64 (the 6809 service requests that provide a similar function). These problems have been corrected.

INTERCEPTS

User signal processing (intercept) routines are now called a little differently. When a signal is received, OS-9 now stacks the mainline program registers on the USER stack instead of the SYSTEM stack. This requires 70 additional bytes of user stack for a signal to be processed. It permits the intercept routine more flexibility; it may restart the main program at an abort recovery entry point instead of continuing where it left off. This also permits intercept routines to exit faster. The F$RTE call may still be used to return from an intercept routine, but it is more efficient to use the equivalent code sequence:

```
movem.l (a7)+,d0-d7/a0-a7   unstack registers
rtr                         return to mainline program
```

I/O CHANGES

The Input/Output manager (IOMAN) has been eliminated as a stand-alone module. Its functions have been incorporated into the OS-9 kernel to provide improved I/O performance.

A new mode byte (M$Mode) has been added to device descriptors. This permits checking for device capabilities and non-sharable devices. It requires all device descriptors to be modified, changing the reserved byte to an appropriate M$Mode value.

A linked list of open paths for each device is now being kept. This requires all version 1.1 device drivers to be at least re-linked with the 1.2 defs before they can be used on 1.2 systems.

RBF CHANGES

The "@" file is now only accessible (beyond the bitmap) by the super user. Non super users reach end of file at the end of the bitmap. Only super users may open it for write access.

The RBF pathlist routines (create, open, delete, makdir and chgdir) didn't always update the caller's pointer correctly. This has been corrected.

If multiple concurrent C compiles were run in the same directory, a bug in RBF record locking routines would sometimes cause the directory structure to become damaged. This has been fixed.

PIPEMAN CHANGES

PIPEMAN has undergone major revision. A major new feature is the support of named pipe files. For complete details, see the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" (chapter 9).

PIPEMAN now supports most getstat/setstat codes commonly used on both RBF and SCF devices. This allows pipelines to be used more transparently with programs expecting a random block device (such as "dir").

The data transfer subroutines used by PIPEMAN are considerably faster now. This makes pipes ideal for applications that require extremely fast communication between concurrent processes.

MISCELLANEOUS KERNEL CHANGES

Several system tables are now automatically expanded if they become full. This permits OS-9 to initially allocate smaller tables, since they are not fixed in size. The tables which will be expanded as necessary are the system module directory, the system path and process tables and the system event table.

MISCELLANEOUS KERNEL CHANGES (cont.)

OS-9 now tries to recover from system state error exceptions. In earlier versions, exceptions such as bus or address trap errors crashed or reset the system. For example, if you tried to access a device whose controller was physically removed from the system, the system was likely to crash. Now, an error is returned when these exceptions occur.

The kernel has hooks to call a user accounting package. In earlier versions, these hooks had bugs that could crash the system. The hooks have been debugged and the user accounting package will be supported in a future release.

DRIVER CHANGES

Serial Descriptors

The "mode" byte was added to the Device Descriptor Header. See Chapter 6 of the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" for details. No driver changes are necessary for this addition.

Serial Drivers (SC7201, SC2661, SC6850)

These drivers have been changed to use the correct register for retrieving a signal code passed by the user when using the SS_SSig setstat call.

There was a bug in the terminate routine which would, under some circumstances, allow the driver to terminate without removing the device from the polling table. This has been corrected.

These drivers have been changed to use jsr instructions instead of a trap instruction when calling the system sleep routine. This results in less system-state overhead.

The way in which baud rate, parity and bits/char is set has been modified/expanded (see Chapter 8 (SCF) in the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" and the utility descriptions of XMODE and TMODE in the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL" for details).

The following Setstats have been added: SS_EnRTS, SS_DsRTS, SS_DCOn and SS_DCOff. These calls are listed in the new manual but are not required for normal operation. They have been included for low level modem operation. Their functions are as follows:

SS_EnRTS: When this call is made to a serial driver, the RTS output on the serial chip should be made TRUE.

SS_DsRTS: When this call is made to a serial driver, the RTS output on the serial chip should be made FALSE.

SS_DCOn: When this call is made, the driver will send a user provided signal code to the process when the DCD line becomes TRUE (i.e. when a modem receives a carrier).

SS_DCOff: When this call is made, the driver will wend a user provided signal code to the process when the DCD line becomes FALSE (i.e. when a modem loses a carrier). NOTE: The signal is only sent on the first transition of DCD and is then disabled. This function is disabled when a setstat SS_Relea is issued to the driver from the process with the same process ID that enabled it.

Serial Driver (SC7201)

To allow for hardware handshaking, the CTS and DCD input lines on the 7201 chip have been enabled. I/O will be inhibited on cards that do not have these inputs in a "TRUE" condition.

The RTS line defaults to the "FALSE" condition upon device initialization. If you would like ths line to default to "TRUE", the following change should be made to the SC7201.a source file.

In the subroutine BitCalc, change the following source statement:

```
ori.b  #DTRLow!TxEnabl,d3
```

to the following:

```
ori.b  #DTRLow!TxEnabl,RTSCntl,d3
```

This will make the RTS line "TRUE" at initialization time.

Disk Descriptors

The "mode" byte was added to the Device Descriptor Header. See Chapter 6 of the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL" for details. No driver changes are necessary for this addition.

Entries have been added for:

Sector Size: this will be used in future releases to define disk sector size. Not implemented in this release but set to 256.

Control Word: Bit 0 has been defined as a format inhibit bit.
1 = inhibit format on this device
0 = enable format
Bits 1 through 7 are reserved for future use for Microware.

Trys: This byte is used to set the number of attempts to read or write a sector. A descriptor with formatting enabled would set this byte to 1 which would in effect disable any retrys.

Disk Drivers (rb765, rb179x, rb8500)

Added code to support the above changes.

The driver now tests for format inhibit in the Write Track routine and also with any attempt to write sector 0. If either of the above is attempted with the format inhibit bit set the driver exits with an error.

It should be noted that Microware has not tested the changes to the rb179x driver due to lack of available hardware.

Clock Drivers (mc6840, am9513 Note: These are not battery backed)

These drivers were modified to return an error if they were passed a Zero for the month. This would prevent anyone from doing a setime with the -s option. (Intended only for battery backup clocks)

Clock Driver (m58167)

The bug which caused the driver to sometimes exit with the stack destroyed is now fixed.

System Debugger

An option has been added in sysinit to inhibit the use of the system debugger.

Ram Disk Driver

These now allow sizes up to 65535 sectors (16 Meg).

MATH CHANGES

"Math1" and "Math2" have been combined into one trap handler called "Math". To use old software that makes TLinks to "Math1" and "Math2", copy the new "Math" module to files named "Math1" and "Math2".

Other Math changes are as follows:

DAdd: The bug causing incorrect results when two operands had equal binary exponents has been corrected.

DtoA: The two numbers passed in register d2 have been changed from bytes to words. Therefore, the "number of digits after the decimal point" may now be greater than 127 or less than -128.

DtoA now works properly when the decimal exponent (result) is zero.

DtoI: Numbers between -1 and 1 would always be rounded to zero. This has been corrected.

DtoL: This did not work with negative numbers with only a fractional part (e.g. -0.723, -0.5). This has been corrected.

DTrn: Now also returns the fractional part (see DtoL above).

Exp: If x/log2 < 1.0, then 1.0 would always be returned. This bug has been corrected.

FtoL: Now also returns the fractional part (see DtoL above).

NumAsc: Numbers greater than 10**308 or less than 10**-308 would return inconsistent results. This has been corrected.

Some number values would cause an infinite loop. This has been corrected.

Misc: The single precision math routines did not function correctly. When registers were restored, d0 was also restored. Because d0 is used for the return value, it would always be destroyed. This has been corrected.

On the 68010, the instruction "move sr,<ea>" is a privileged instruction. Some math routines used this instruction and because it was executed in user mode, did not function properly. This has been corrected.

Misc: Numbers which are exactly between two whole numbers (e.g. 30.5 or 17.5) are now properly rounded. IEEE specifies that numbers of this type should always be rounded to the even number.

Fixed rounding of numbers where abs(x) < 0.5.

Many functions have been rewritten with major speed and size optimizations.

UTILITY CHANGES

Attr: A setstat was being done on a file even when no attributes were being changed, which could cause an error if you didn't have permissions to change it. The setstat was removed in that case.

Checks were installed to prevent conflicting options. i.e. "attr -prnpr". Also the syntax of the options was tightened up to prevent "attr -nar", etc. This was interpreted as "-a -nr".

The error message generated when you try to turn off the "d" bit and the directory is not empty has been changed to "directory not empty".

The syntax message in the command summary (-? option) has been changed slightly to reflect the true syntax. The brackets were in the wrong place.

Binex: An -a option has been added to specify the load address in Hexadecimal.

S2 and S3 records can now be generated. (see the Binex utility description in the OS-9/6800 OPERATING SYSTEM USER'S MANUAL for details).

Cfp: Cfp now executes the procedure file by piping it to the shell to insure deletion of the temp file even when the cfp is aborted.

An error message is now generated, if no procedure file is given and the -s option is not used.

Cmp: The syntax for -s option, in the command summary (-? option), has been changed to conform with the manual description of the option.

Copy: The 218 error generated when copying to an SCF device has been fixed.

The "continue ?" message now requires a "y" or "n" as a response.

Date: The "-t" option is now transparent. Date will now always print the time.

Checking for conflicting options (e.g. -m and -j) has been improved.

Dcheck: The "-?" option has been removed from the help message.

Del: New prompt options have been added when using the "-p" option: Delete ? (y,n,a,q).

Deldir: Only 1 character is now read after the (d,l,q) prompt. Previously, a 102 error could have occurred if too many characters were entered and the input buffer was overrun.

A clearer prompt message is now generated after listing a directory: "delete ? (y,n)". Deldir now allows you to skip a directory and continue deleting other directories.

Dir: Dir -e no longer uses the open of @. Instead, special system calls are issued to locate the individual file's FD sector. The open of @ is now a privileged function.

Dir now sorts the input names.

The -a option has been added. This allows file names beginning with "." to be displayed. Without the -a option they will not be displayed by "dir".

Dir now prints all files first and then directories when using wildcards from the command line.

The -d option has been added to append "/" on directory names for easy identification of directory files in output.

Dsave: The -e option has been added to immediately execute its output.

The -r option has been added for rewriting over files (effectively uses copy with -r option).

The -d option has been added to copy files by date when doing selective backups. This option copies only files with newer dates.

The -s option has been added to circumvent the "Continue ? (y/n)" prompt on error. When using -s, dsave will skip the file (that caused the error) and keep going.

The -a option has been added to make dsave sensitive to file names beginning with a "." (period). The -a option causes dsave to ignore these files.

The continue prompt has been changed to "continue (y,n,a,q)".

Dump: The last line was being shown as a duplicate even when it had fewer than 16 characters on it. This has been corrected.

The address bug when dump started in mid-file has been fixed. It used to display starting point as 0 regardless of where you started.

A message is now generated if no input was read.

The -d option has been removed from help message.

Echo: The inconsistencies with receiving arguments from the command line and from standard input have been corrected.

Expand: Expand now closes the input file before deleting it. This caused a 253 error.

The "-nz=file" combination no longer produces a #103 error.

Ident: Ident no longer lists a system module as an unknown type.

The module type and language are now printed as text.

Load: The -d option has been added to load a module from your current directory instead of just from the execution directory.

Login: Login assumed a "null password" when an extra space was typed at the end of a user name. This has been corrected.

If you had multiple entries in the password file with the same user name and different passwords, and one entry had a null password, you could not log on with a name and password that was listed after the "null password" entry. This has been corrected.

Previously, a "de-reference through zero" bug caused bus trap errors on memory protected systems. This has been corrected.

Make: The RDIR macro tested the character pointed at by a null pointer instead of the pointer itself being null. This has been corrected.

RDIR, SDIR and ODIR were improperly appended on command lines. This has been corrected.

Make (cont.): Make now correctly handles a tab between names before a colon on the command lines.

The spacing in the error message, "can't open makefile", has been corrected.

Make now forks a process with the same priority as itself, rather than always 128.

Mdir: The -u option has been added for unformatted listings.

Wildcard searching and selective requests for individual modules have been added. Wildcards must appear in quotes, otherwise the Shell will expand them.

The -t option has been added for displaying only certain types of modules.

Mdir information is now displayed in two formats. "Mdir" prints the type field and "Mdir -a" prints the language field.

The type/language information is now using mnemonics. See Mdir utility description in the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL" for details.

Pd: Previously, the option parser only checked the first option specified on the command line. This has been corrected.

Pr: The -z option now correctly prints the title.

When using the -m option, the title will be omitted unless specifically set using the -u option.

Using the -n option twice would increment line numbers by two each time (using it three times would increment line numbers by three). This has been corrected.

Procs: If the system clock was not started, procs would terminate with a 105 (divide by zero) error. This has been corrected.

The "grp/user" field is now displayed in the default procs display and the "aging" field is now displayed in the alternate format.

If both the -b and -a options are specified, only the -b option is used.

Qsort: Specifying zero (0) in the -f option is now illegal.

Rename: Filenames are now allowed to start with a '.' (period).

Some error messages have been changed for clarity.

Save: When using both -f and -z=file options, save now behaves as expected.

Setime: The -s option has been added for battery backed up clocks.

Arguments on the command line may now be separated by spaces.

Setime will automatically call the date utility to display the current time. The -d option prevents this.

Checks for illegal days within a month (i.e. Feb 29) have been added.

An am/pm option has been added. The prompt and the syntax message has been changed to reflect this addition.

Shell: Arguments are now passed to the process with the C language function, os9exec(). This allows more flexability in argument passing.

When forking a Shell by way of parenthesis (e.g. $ (chd)), the number of trailing parentheses was not checked. This has been corrected.

Setenv, printenv and unsetenv commands have been added to handle environment variables.

The Set command has been added to set Shell parameters. Parameters may still be set by listing only the option on a command line.

Tmode: The baud rate field has been added.

A user is now able to change parity, character size and number of stop bits with new mnemonics. These are combined for display in the type byte.

Xmode: The baud rate field has been added.

A user is now able to change parity, character size and number of stop bits with new mnemonics. These are combined for display in the type byte.

SCRED CHANGES

The "scr" prompt has been changed to ">".

In Version 1.1, typing a linefeed in the middle of a line would cause the cursor to be shown in an incorrect position. This has been corrected. The displayed position now reflects the true position of the cursor.

Scred now pre-extends the output file in order to prevent disk fragmentation.

Scrolling routines have been optimized to reduce the amount of I/O processing.

After using the "more" command with a long file, and then searching for a non-existant string, Scred now returns the cursor to its original position.

New buffer extension and compression routines have been written in assembly language for improved performance.

A "-g" option has been added for terminals with no linefeed.

The "k" key (in Edit mode) now moves the cursor to the end of the line if it is already positioned at the beginning of the line (and vise versa).

The cursor is now positioned at the beginning of the new text, after inserting a text file with the "add" command.

The status line is now displayed only upon request.

The cursor is now returned to its proper position when returning to Edit mode from Command mode.

The cursor addressing mode and the cursor position offset are now seperate in the termset file.

The termset file now has screen length and screen width fields.

Better error messages have been added for invalid buffer size requests.

The goto command (in Command mode) now reprints the screen and correctly positions the cursor when specifying a line on the page you were last editing.

Pasting text directly after inserting characters on a line no longer corrupts the file.

After writing a marked section, highlighting of sections/lines is now correctly turned off.

Error messages are now returned for using old or incomplete termset files.

Scred now allows larger files to be added using the "add" command. The size of the file to be added may now fill the entire non-used space of the edit buffer:

Edit_buffer - text - cut_buffer = space_to_add_text

Scred now uses a getstat for file size to insure this qualifications before adding any text. If the file is too large, an error message is returned and no text is added.

An error check has been added and appropriate error messages are returned (upon error) when Scred renames "ed.temp.xxx" files.

Getenv() is now used to check the environment variable "TERM" for the type of terminal to be used. If the "-t=term" option is used, it has the greatest priority. Next the environment is checked for "TERM". If neither of these are used, the default terminal type is assumed.

Getenv() is also used to check the environment variable "DDEV" for the default drive to search for the termset file. If "DDEV" is used, the specified device is searched before the usual checking sequence (/dd/sys, /h0/sys, /d0/sys).

Deleting the first line of a file no longer causes the file to be displayed incorrectly and no longer causes any textual problems within the file.

Error numbers are now displayed with error messages.

BASIC09 CHANGES

After any procedure is run, the first procedure in the directory no longer becomes the current procedure.

Typing a "save*" without giving a file name no longer causes the standard input path (#0) to get closed, resulting in an endless "What?" loop.

BASIC09 programs can now open files in the execution directory, i.e., "OPEN #path,"file":READ+EXEC".

SUBSTR function now works correctly when it finds a partially matching string immediately prior to a fully matching string.

When DEBUG mode is entered because of error #047 and a user executes a list command, BASIC09 no longer crashes.

BASIC09 system mode now reads commands from standard input.

When both internal and external procedures exist in the directory, and an internal procedure is edited, then a DIR is attempted, BASIC09 will no longer crash.

With the "r" format in PRINT USING, the correct number of digits for numbers that are less than 0.1 are now printed.

When stepping through a procedure in edit mode, bottom-of-loop statements are no longer indented.

The decompiler's problem of putting too many parentheses around expressions which contain functions has been corrected.

When "auto running" a packed program, BASIC09 now does an F$EXIT when the program terminates.

If "TRACE" is on, and another procedure is run, both the statement decompilation and expression trace are now turned off.

If an executable statement follows a non-executable statement on the same line, the executable statement now is executed.

With a statement such as "i = j \ j = 1 \ 1 = i", TRACE mode now prints each statement seperately prior to its execution:

```
i = j
j = 1
1 = i
```

The BASIC09/68K compiler now restricts the arguments of the ADDR and SIZE functions. In previous versions it was allowing expressions rather than just variables.

The expression trace is now disabled after entry to debug mode.

The CREATE and OPEN routines in the interpreter now skips statement terminators (<CR> or "\"). This was a problem, because a terminator would get printed out twice when trace mode is active.

Using BASIC09's list feature and redirecting the output to /p1 (a serial printer) no longer causes the offset addresses to be sent to the terminal and the lines of code to be sent to the printer.

The change command in the editor can now make changes to extremely long lines (greater than 100 characters).

Line numter problems (i.e. typing a line number in edit mode always going to the first line in the file) have been corrected.

Procedures are now bound correctly even if the available space is greater than 64K.

The BASIC09 INPUT statement will now allow hex values to be entered by preceding the value with a "$".

The "Val" function now does conversions on hex values (i.e. values with a "$" at the beginning).

When using the H format for INTEGER, BYTE and BOOLEAN values with PRINT USING, BASIC09 no longer prints the high order bytes when the specified field width is not large enough.

The PRINT USING format "R8.2>" and "R10.2^" no longer puts the minus sign on the wrong side of the number.

BASIC09 no longer closes any open paths when normally returning to command mode.

The sequence of renaming a non-current procedure and then listing, packing or saving the procedure by name no longer crashes the system.

The LIST command from DEBUG mode now denotes the current line (*) if the line had more than one statement and the I-code pointer points to a statement other than the first. For example:

```
* i=j \ x=y \ PRINT x,i
    ^
  I-code pointer
```

BASIC09 does now sets the user and group ID when it packs a procedure.

Running an empty procedure no longer crashes BASIC09.

An error is now reported if an error occurs during the tracing of a statement.

The RUN command now correctly saves procedure addresses. It no longer searches the directory each time a procedure is run.

Several problems have been corrected that had caused the system to crash:

"pack proc":	when proc is already packed in the workspace.
"rename proc1 proc2":	when proc1 is packed and in the workspace.

The PACK procedure no longer removes end-of-line tokens at the end of CREATE and OPEN statements. This caused strange problems for any programs being run as packed procedures.

The INPUT statement now validates its input variables.

When in edit mode, a "d+*" now correctly displays the current line after deleting to end of file.

When inserting a numbered line with the BASIC09 editor it will now insert the line at the current edit pointer position if sequentially correct. Otherwise the line will be placed directly before the line with the next higher number.

The amount of available memory is now checked correctly when a procedure is interpreted.

Listing a procedure that has a nesting deeper than 30 levels no longer causes BASIC09 to hang up; an error message is returned instead and the listing is terminated.

When BASIC09/68K executes the command "CHAIN "ex runb"", it now correctly unlinks all external procedures and returns memory to the system.

The BASIC09 debugger no longer allows the assignment or printing of structured data type variables. If either is tried, an error is generated and the instruction is not performed.

If the carry is set when a machine language module (being executed from a BASIC09 procedure) exits, the correct error code is now returned and Debug mode is entered

2 and 3 dimensional arrays now work.

Naming conventions for procedure names now correspond to OS-9 naming conventions. This includes the characters allowed in names and the length of names.

C COMPILER CHANGES

The following is a list of the known and corrected problems in the Microware OS-9/68000 C Compiler System. These fixes apply to both the resident and cross versions of the software. The number in square brackets [] indicates the edition number to which the bug fix applies.

cc: (executive) cc68 for cross compiler systems

"cc" has been changed to link with math.l, because the math libraries have been combined into one file. In addition, the -h option is no longer available because there is only one math library to choose from. [ed12]

The -j option that causes the linker to generate a jumptable for distant code references is now enabled by default. The -j option now does NOT cause the linker generate a jumptable. The effect of the corresponding option in the linker (-a) has not been changed. [ed13]

The -v= option can appear more than once to specify additional directories for the pre-processor to search when handling the #include <> directive. The directories are searched in the order given. Additionally, if no -v= appears the directory "/dd/defs" is searched. If that directory does not exist, the environment is checked for the variable DDEV which indicated the device on which to look for the "defs" directory. [ed14]

cpp: (C preprocessor)

Sometimes cpp tried to call detach() with -1 as the pointer causing some versions of the kernel to crash in system state. This only affects resident 68000 versions. [ed6]

A "-o=" option has been added to cpp to avoid re-direction from cc68.

Embedded assembly lines are not passed through as comments when embedding C source lines in assembly output. This eliminates the assembly lines being listed as a comment lines. [ed8]

The -v= option can appear more than once to specify additional directories for the pre-processor to search when handling the #include <> directive. The directories are searched in the order given. [ed9]

c68: (68000 C Compiler)

Code generated for long constant pushes to the stack has been improved. For constants in the range -32768 <= n <= 32767, a "pea <n>.w" is generated instead of "move.l #<n>,-(sp)". Constants outside this range still use the latter form. Also, address register moves to the stack are accomplished by: "pea (an)" instead of "move.l an,-(sp)". [ed13]

Signed short quantities are pushed to the stack as:

```
movea.w <ea>,a0
pea (a0)
```

The movea.w automatically sign-extends the source word to a long. The pea (a0) is a quick way to get the contents of an a-register to the stack. [ed13]

The ++ operator applied to a "float" operand didn't work because T$DInc was called instead of T$FInc. This has been corrected. [ed13]

An indexing bug involving certain array subscript calculations has been corrected. For example, given the following:

```
struct joe {
char xxx[4],yyy[4];
} lax[16];

...

lax[i].yyy[j] = '0';

...
```

The offset (4) from the "lea" of the inner subscript would "stick" to the expression and cause a "move.b #'0',4(a0,d0.l)" instead of the correct "move.b #'0',0(a0,d0.l)". The offset value is now cleared when an indirection (lea nnn,a0) is performed. [ed14]

Assignment operators involving float types did not work. Sometimes incorrect code was output. At other times the compiler would quit with a "dreg free" fatal error message. This has been corrected. [ed15]

o68 (cont.):

Alignment is now always enforced for "unsigned short" and "unsigned char" types. Global and static structs are now aligned on word boundaries (i.e., ds.w). [ed16]

Division by powers of 2 for signed numbers are now performed by calling the divide routine instead of arithmetic shifts. Shifting negative quantities is not correct because the value approaches -1 rather than zero. [ed17]

A type is now required in the struct-declaration of a struct-decl-list. For example, in the following declaration, the type-specifier is missing from the "moe" member struct-declaration:

```
struct xxx {
int i;
char j;
moe;
};
```

A "type required" error is now issued when this is detected. [ed19]

Register utilization involving certain integer multiply, integer divide, float and double expressions is now improved. [ed20]

Typedef involving arrays and pointers now work:

```
typedef char *xxx;
xxx func();
```

The type of "func" should be "function returning pointer to char". This used to evaluate to "pointer to function returning char". [ed21]

Types have been expanded to allow 14 levels of indirection rather than 6 with a 16-bit type specifier. [ed22]

Previously, certain type conversions did not work properly: unsigned types to most other types, and float and double types to short or unsigned short (char). This has been corrected. [ed23]

c68 (cont.):

Sometimes array accesses would generate incorrectly formed assembler instructions. For example:

These instructions:

```
int index,ptab[10][40];
main()
{
ptab[index][32] = 20;
}
```

would fail when assembled with:

```
*** error - bad operand ***
      addi.l #128, :6
```

This strategy for array accessing has been improved to avoid this problem. [ed24]

The sizeof operator would not return the correct type for de-referenced arrays. In particular,

```
int i[10][4][8];
```

expression:	returned:	instead of:
sizeof i	1600	1600
sizeof i[x]	4	160
sizeof i[x][y]	4	32
sizeof i[x][y][z]	4	4

sizeof now returns the proper size for arrays. [ed25]

The >>= operator was not generating proper code for various classes of unsigned lvalues. [ed26]

Boolean tests involving short quantities and large constants >32k were not properly handled. [ed27]

The code for boolean tests against zero and powers of two has been improved. [ed28]

The code for certain operations involving short data types has been improved to yield shorter and faster code. [ed29]

Dimension information for declarations like:

```
int (*(*(*(*(*w[10])()) [8])()) [5])();
```

are now handled properly. The problem with handling constant subscripts has been fixed, also. [ed30]

c68 (cont.):

Code generated for switch statements has been improved, resulting in shorter and/or faster case selection. [ed31]

Types "unsigned long", "unsigned short", etc. are now allowed on struct member names. [ed32]

If a function declared its first 3 arguments as an int and the fourth as a double, the double argument's address would be the same as the last int. [ed33]

The "remote" storage class has been added for external variables to allow huge arrays to be statically declared. [ed34]

Negative float/double values are now accepted in initializers. [ed35]

Inc/dec operators applied to double types now generate the correct math library call. [ed36]

Assignment operators used with array destinations would sometimes foul up the temporary register used for the array destination. This has been corrected. [ed38]

The addr/bus trap problem involving declaration of arrays of structs has been fixed. [ed39]

The kanji-version compiler now handles the kanji code sequence properly if \n appears in the string. [ed40]

The auto increment/decrement addressing mode is now being used when accessing pointers to pointers. [ed41]

The format of the error message line has been changed slightly to allow filter programs easy access to error messages. The line now looks like:

"file.c", line 1: **** error message ****

The erroneous line is still listed after the error message with a pointer arrow to the offending element. [ed42]

Incorrect code was occasionally generated if the first argument to a function was a double expression cast to an int. This has been corrected. [ed43]

c68 (cont.):

Under certain conditions, comments in the assembly listing would appear immediately after the compiler-generated labels causing assembler syntax errors. This no longer happens. [ed44]

The compiler line buffer size has been increased from 256 bytes to 512 bytes. The maximum length of an input line to the compiler is 512 bytes. A string or preprocessor macro continued over more than one physical line is limited to this 512-byte limit. [ed45]

The exponent range check to limit exponents on double constants to the range of -308 <= exp <= 307 has been fixed. [ed45]

Code generation for expressions involving remote double variables has been improved. [ed46]

The form feed character (\f) and the vertical tab character (\v) is now interpreted as white space. [ed47]

o68: (C object-code improver)

The current edition is [ed8].

r68: (68000 Macro Assembler)

The problem with the "equ" directive involving global symbols has been fixed. For example:

```
        psect
joe:    equ *
        nop
        ends
```

The label "joe" previously was not associated with the code area by the assembler thereby causing the linker to treat references to "joe" as absolute address access. Since this change affects the format of the assembler output, the linker will interpret output file edition 4 as having this change, while all previous versions being interpreted as the old style. This is so library (and previously assembled files) do not have to change.

The macro work file is now qualified by process id so more than one r68 can be run in the same directory. [ed12]

r68 (cont.):

For ease of portability to and from other assemblers, r68 now allows a size extension on machine mnemonics that have no choice of size. For example:

" seq xx" can appear as " seq.b xx"

The memory shift instructions were flagged with "illegal size" errors if the size was not byte. This has been fixed to happen only if the size is not word. [Ed13]

The "move a0,usp" instruction no longer causes r68 to fail with an internal assembler error: group1a ffff8000. [ed14]

Support for "remote" vsects has been added. This also required a change to the rof. The rof number is now 5. [ed15]

The offset for indexed register indirect with offset addressing mode "n(a0,d0)" is now restricted to the range -128 >= x <= 127. [ed16]

A -? option has been added to display the r68 command line and synopsis of the options. [ed17]

168: (OS-9/68000 Linker)

Psects larger that 32k previously would cause the linker to foul up the reference resolution for offsets greater that 32676. The linker will now handle both new (ed 4) and old (pre-ed 4) assembler output files. The warning:

"168: warning - '<filename>' should be re-assembled"

appears when an old-style input file is read. The linker can still process these files. If an old-style input file is read, and a psect within it is larger than 32k, this warning turns into an error. The file must be reassembled with the new assembler before linking. [ed11]

The bug in "equ" that caused a constant value to be biased by the base of the data when the "equ" appeared in a vsect has been fixed.

A "-p" option has been added to set permission in the module header. The global label "_sysperm" is now recognized to allow module permission to be set.

168 (cont.):

Error messages for duplicate name problems has been improved.

Jumptable information is now written to stdout. [ed13]

Allowable psect and file name length has been icreased from 32 to 255 characters. [ed14]

Support for the "remote" data area has been added. The remote area is an extension of the static storage beyond the 64k absolute limit. This allows huge arrays to be allocated with the C directive "remote". This is the inverse of the 6809 direct page. The 68k can only address 64k bytes with the "register indirect with offset" addressing mode. The compiler generates instructions to offset the data pointer with a large constant value, thereby providing a full 32-bits of index. The linker separates the vsects indicating "remote" and allocates all the remote storage after the normal static and data areas. [ed12]

Program-counter relative lea instructions that reference destinations farther away than 32k are now converted to movea instructions that extract the destination address from the jumptable. [ed15]

A -? option has been added to display the r68 command line and synopsis of the options. [ed16]

An error is now reported if initialized data is allocated to modules other than "program" or "trap handler" modules. [ed17]

Various linker error messages have been improved. [ed18]

The resident 68k linker will place symbol modules in a directory named STB in the execution directory, if STB is present, otherwise, the symbol module is left in the execution directory. This is to keep the execution directory as small as possible. Cross versions of the linker leave the symbol file in the same directory as the output file. [ed19]

cio/clib:

Cio can now be used with programs larger than 32k.

Scanf() no longer returns incorrect values for integer quantities.

cio/clib (cont.):

Fclose() previously failed to close a file open for write access if no writes were performed to the file. This could result in running out of path numbers for the process. This has been corrected.

If pflinit() or pffinit() are called in a program being linked against the cio traphandler (cc68 -i), the linker no longer complains about duplicate symbol definitions for printf(), fprintf() and sprintf().

The documentation for modlink() and modload() was incorrect. The proper arguments for these functions are:

```
mod_exec *modlink(modname,typelang)
char *modname;
short typelang;

mod_exec *modload(modname,accessmode)
char *modname;
short accessmode;
```

"typelang" indicates the desired type and language for the module being linked. "accessmode" is the mode with which to open the file to load.

The printf() %g conversion now suppresses trailing zeroes after the decimal. Additionally, the decimal point is not displayed if the number has no fraction.

The type/language parameters for os9fork() and chain() library calls are now being passed to the system correctly.

The sbrk() function is now rounding the given size to an even bytecount.

The rindex() function previously would not find the specified character if the only occurrance of the character happened to be the first character of the given string. This has been corrected.

The _srqmem() function no longer fouls up the stack if the system could not grant the memory size requested.

cio/clib (cont.):

The following functions have been added to provide access to contents of OS-9 directories in a more machine-independent manner. These functions operate like the BSD4.2 functions of the same name:

```
closedir()- close a directory file
opendir()- open a directory file
readdir()- read the next directory entry
rewinddir()- return to the beginning of the directory
seekdir()- move to position in directory
telldir()- report position in directory

<dir.h>- defines structures for accessing directories
```

It is now possible to pass an environment to a C program. Also, the method in which the arguments are passed has been improved by allowing the entire "argv" pointer list to be passed to the new process. This facility requires linking existing programs with the new "cstart.r" module, which, handles the "old" method of argument passing from os9fork() as well as the new method provided by os9exec(). cstart will call the main() function of the program as:

```
main(argc,argv,envp)
int argv;
char *argv[], *envp[];
```

The argc and argv are received in the traditional manner. argv[0] points to the name of the running module. "envp" is an array of pointers to the environment strings provided by the parent process. The list of pointers for both "argv" and "envp" are terminated by a NULL (0) pointer.

Typically an environment variable as set by the shell looks like:

```
USER=joe
```

The name of the variable is "USER" and the value of "USER" is "joe". The case of the letters in the variable need not be uppercase, but case is significant.

The startup routine "cstart.r" will now TLink to the math trap handler module called "math" for both trap 14 and trap 15 traps. Previously, trap 15 would TLink to "math1" and trap14 would TLink to "math2".

cio/clib (cont.):

A new function called getenv() can be used to access environment variables by name:

```
char *getenv(name)
char *name;
```

"name" is a pointer to a character string representing the name of the desired variable. If the variable is present in the environment, getenv() returns a pointer to the value string. If the variable is not in the environment, a NULL (0) pointer is returned.

A new function called os9exec() has been added to the library to allow a new process to access the new argument and environment facility:

```
#include <os9exec.h>

os9exec(func,modname,argv,envp,datasize,priority,pathcnt)
int (*func)();/* usually os9forkc() */
char *modname;
unsigned datasize;
short priority,pathcnt;
char **argv, **envp;
```

"modname" is the name of the module to fork or chain. "datasize" is the additional amount of stack memory for the new process; zero for no change. "priority" is the priority for the new process; zero for no change. "pathcnt" is the number of open paths to pass on to the new process. "argv" is an array of character pointers pointing to the argument strings for the new process. The last pointer in the array must be a NULL (0) pointer. "envp" is an array of character pointers pointing to the environment strings for the new process. Normally, the external global variable "environ" is passed here to cause the current environment to be made available to the new process. "func" is a pointer to the function that will perform a fork or chain.
The arguments for this function are the same as for os9fork(). Generally, the functions os9fork(), chain(), os9forkc() or chainc() are used. os9forkc() and chainc() are used if the "pathcnt" value is to be used. When using os9fork() or chain(), "pathcnt" is not required. The value returned by the "func" function is returned by os9exec().

clo/clib (cont.):

The C startup routine (cstart.r) has been changed to recognize the environment. If a program is forked via the os9exec() function, cstart.r use the pointer offset information passed from os9exec() to re-construct the argv pointers to the argument strings, and will make the environment strings available to the program via the "envp" argument to main, and by the global variable "environ". Programs forked by the old os9fork() function cannot receive the environment, and cstart.r will parse the parameter string for the arguments.

Two new functions have been added to the library to provide access to the F$Fork and F$Chain system call parameter that indicates the number of open paths for the child process to inherit:

```
os9forkc(modname,parmsize,parmptr,type,lang,datasize,prior,
        pathcnt)
char *modname, *parmptr;
int parmsize, datasize;
short type, lang, prior, pathcnt;

chainc(modname,parmsize,parmptr,type,lang,datasize,prior,
      pathcnt)
char *modname, *parmptr;
int parmsize, datasize;
short type, lang, prior, pathcnt;
```

The arguments are defined as for os9fork() and chain(). The pathcnt is the number of open paths for the new process to inherit. Normally, this value is 3 to cause at least standard input, output and error to be inherited. os9forkc() returns the process id of the child process.
chainc() does not return, unless the chain was unsuccessful, in which case it returns the system error number.

Standard library functions that read stdin will now do readln() calls by default regardless of the type of the device. Previously, when reading stdin from an RBF-style device, the routines would use read() which caused buffering problems when forking processes that continue to read the parent's standard input path.

The startup routine "cstart.r" now defines a global short value (_pathcnt) that contains the number of paths open when the process was created.

MICROWARE SYSTEMS CORPORATION

DES MOINES