OS-9® for 68K Processors Technical Manual

Version 3.0

MICROWARE™

Intelligent Products For A Smarter World

Copyright and Publication Information

This manual reflects version 3.0 of OS-9 for 68K Processors Technical Manual.

Revision: D
Publication date: September 2000

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, Microware will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction Notice

The software described in this document is intended to be used on a single computer system. Microware expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of Microware and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

For additional copies of this software/documentation, or if you have questions concerning the above notice, please contact your OS-9 supplier.

Trademarks

Address

Microware Systems Corporation
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Table of Contents

Chapter 1: System Overview

This chapter provides a general overview of OS-9's four levels of modularity, I/O processing, memory modules, and program modules. It includes the following topics:

- **System Modularity**
- **I/O Overview**
- **Memory Modules**

System Modularity

OS-9® has four levels of modularity:

1. **The Kernel, IOMan, the Clock, and the Init Modules**
   The kernel provides basic system services including process control and resource management. IOMan provides Input/Output (I/O) management, The clock module is a software handler for the specific real-time-clock hardware. The Init module is the initialization table the kernel uses during system startup.
2. **File Managers**
   File managers process I/O requests for similar classes of I/O devices. Refer to the I/O Overview in this chapter for a list of the file managers Microware currently supports.
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, serial port drivers for terminals and serial printers, and a driver for parallel printers. You can also add customized drivers of your own design or purchase drivers from a hardware vendor.
4. **Device Descriptors**
   Device descriptors are small tables associating 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 using 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.

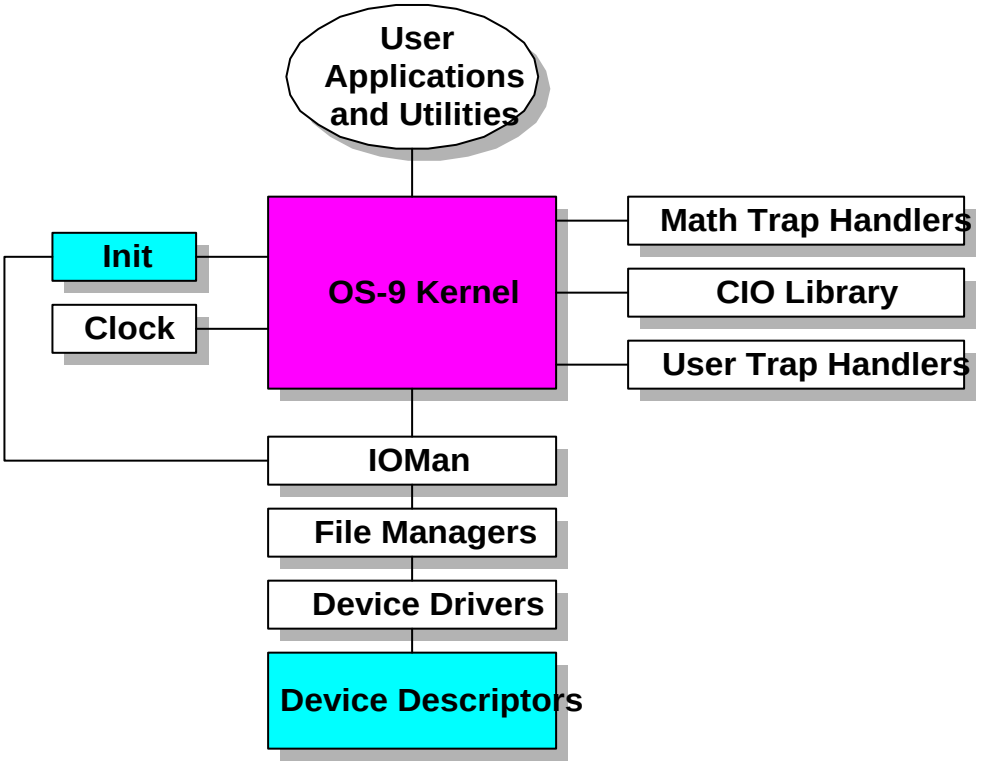
For More Information

For specific information about file managers, device drivers, and device descriptors, refer to:

- Chapter 1: System Overview (this chapter)
- Chapter 3: OS-9 Input/Output System

- The ***OS-9 for 68K Processors Technical I/O Manual***

Figure 1-1 OS-9 Module Organization



Note

The shaded boxes contain non-executable code. These modules are referenced, not *called*. The kernel, file managers, and device drivers reference descriptors directly, but only the kernel and IOMan reference the `Init` module directly.

An important component, the command interpreter (the shell), is not shown in this diagram. The shell is an application program, not part of the operating system. It is described fully in ***Using OS-9 for 68K Processors***. To obtain a list of the specific modules making up OS-9 for your system, use the `ident` utility on the `OS9Boot` file.

Although all modules could be resident in ROM, the system bootstrap module is usually the only ROMed module in disk-based systems. All other modules are loaded into RAM during system startup.

I/O Overview

IOMan maintains the I/O system for OS-9. The kernel provides the first level of I/O service by routing system call requests between processes and IOMan, which then passes the requests to the appropriate file managers and device drivers. Microware includes the following file managers in the OS-9 for 68K distribution.

Embedded

Table 1-1 Embedded File Managers

Name	Description
SCF	**Sequential Character File Manager** Handles I/O for sequentially character-structured devices, such as terminals, printers, and modems.
PIPEMAN	**Pipe File Manager** Supports interprocess communications through memory buffers called *pipes*.

Disk Based

Table 1-2 Disk Based File Managers

Name	Description
RBF	**Random Block File Manager** Handles I/O for random-access, block-structured devices, such as floppy/hard disk systems.
SBF	**Sequential Block File Manager** Handles I/O for sequentially block-structured devices, such as tape systems.
PCF	**PC File Manager** Handles reading/writing PC-DOS disks. It uses RBF drivers and is sold separately.

Extended

Table 1-3 Extended File Managers

Name	Description
IFMAN	**Communications Interface File Manager** Manages network interfaces.
PKMAN	**Pseudo-Keyboard File Manager** Provides an interface to the driver side of SCF to enable the software to emulate a terminal.

Table 1-3 Extended File Managers (continued)

Name	Description
SOCKMAN	**Socket File Manager** Creates and manages the interface to communication protocols (sockets).
NFS	**Network File System Manager** Client file manager for mounting remote file systems. The NFS protocol provides remote access to shared file systems over local area networks.

Other

Microware also supports the following file managers that are not included in the standard OS-9 distribution:

Table 1-4 Other Microware File Managers

Name	Description
CDFM	**Compact Disc File Manager** Handles CD and audio devices, as well as access to CD ROM and CD audio.
UCM	**User Communications Manager** Handles video, pointer, and keyboard devices for CD-I (Compact Disc-Interactive).
GFM	**Graphics File Manager** Provides a full set of text and graphics primitives, input handling for keyboards and pointers, and high level features for handling user interaction in a real-time, multitasking environment.

Table 1-4 Other Microware File Managers (continued)

Name	Description
NFM	**Network File Manager** Processes data requests over the OS-9 network.
NRF	**Non-Volatile RAM File Manager** Controls non-volatile RAM and handles a flat (non-hierarchical) directory structure.
ISM	**ISDN Basic Rate Interface Manager** Manager connections for Basic Rate (2B+D) Interfaces to the Integrated Services Digital Network (ISDN).

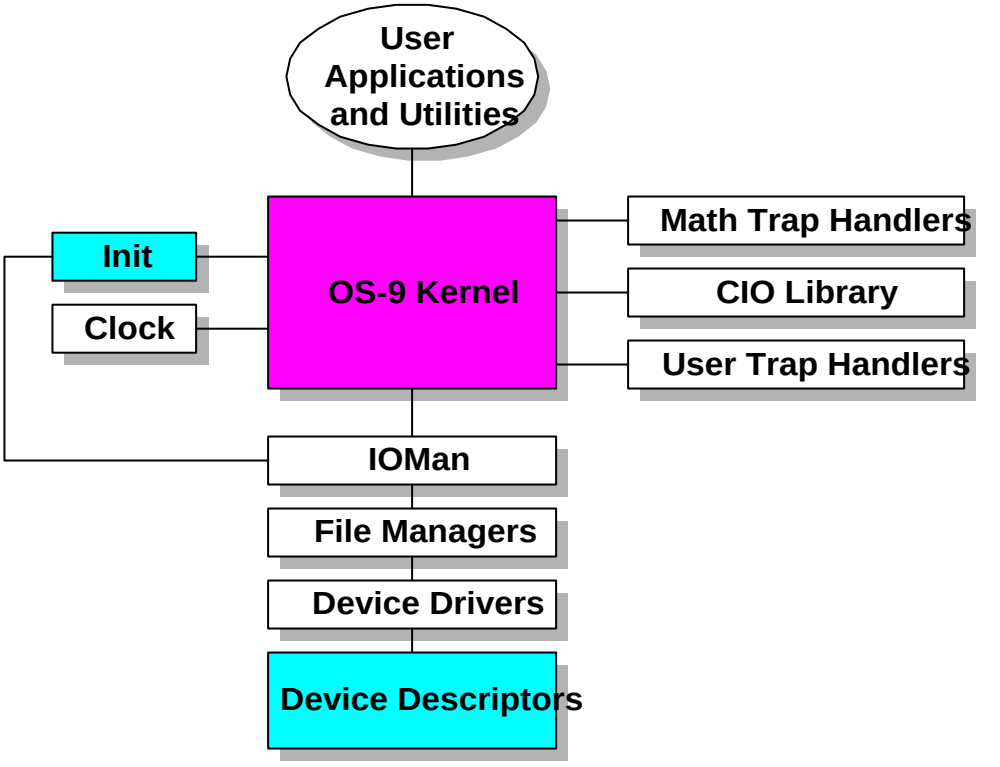
For More Information

For specific information about these file managers, refer to:

- Chapter 4: Interprocess Communications
- The ***OS-9 for 68K Processors Technical I/O Manual***

Figure 1-2 illustrates how OS-9 processes an I/O request:

Figure 1-2 OS-9 Module Organization



Note

The shaded boxes contain non-executable code. These modules are referenced, not ***called***. The kernel, file managers, and device drivers reference descriptors directly, but only the kernel and IOMan reference the `Init` module directly.

Memory Modules

OS-9 is unique as it uses *memory modules* to manage both the physical assignment of memory to programs and the logical contents of memory. A memory module is a logical, self-contained program, program segment, or collection of data.

OS-9 supports ten pre-defined types of modules and allows you to define your own module types. Each module type has a different function.

The ten pre-defined memory types are defined by `M$Type` in the module header definition as follows:

`Prgm`	Program module
`Sbrtn`	Subroutine module
`Multi`	Reserved
`Data`	Data module
`CSDData`	Configuration status descriptor
`TrapLib`	User trap library
`Systm`	System module
`Flmgr`	File manager module
`Drivr`	Physical device driver

Table 1-5 Memory Module Characteristics

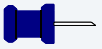
Modules do not have to be	Modules are required to be
Complete programs.	Re-entrant.

Table 1-5 Memory Module Characteristics (continued)

Modules do not have to be	Modules are required to be
Written in machine language.	Position-independent. Conforms with the basic module structure described in the next section.

The 68000 instruction set supports a programming style called *re-entrant* code—code that does not modify itself. This allows two or more different processes to share one ***copy*** of a module simultaneously. The processes do not affect each other, provided each process has an independent area for its variables.

Almost all OS-9 family software is re-entrant, and therefore uses memory very efficiently. For example, `umacs` requires 41K bytes of memory to load. If you make a request to run `umacs` while another user (process) is running it, OS-9 allows both processes to share the same copy, saving 41K of memory.



Note

Data modules are an exception to the re-entrant requirement. However, careful coordination is required for several processes to update a shared data module simultaneously.

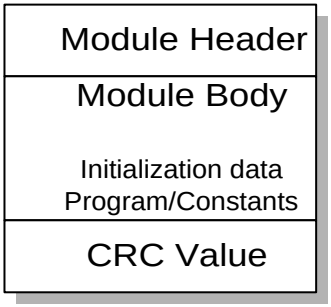
It does not matter where a *position-independent* module is loaded in memory. This allows OS-9 to load the program wherever memory space is available. In many operating systems, you must specify a *load address* to place the program in memory. OS-9 determines an appropriate load address for you when the program is run.

OS-9 compilers and interpreters automatically generate position-independent code. In assembly language programming, however, the programmer must insure position-independence by avoiding absolute address modes.

Basic Module Structure

Each module has three parts: a *module header*, a *module body*, and a *CRC value*.

Figure 1-3 Basic Memory Module Format



For More Information

For specific information about the structure and individual module header fields, refer to the list at the end of this chapter.

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

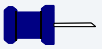
The module body contains initialization data, program instructions, and constant tables.

The last three bytes of the module hold a CRC value (Cyclic Redundancy Check value) to verify the module's integrity. The linker creates the CRC at link-time.

The CRC Value

The CRC (Cyclic Redundancy Check) is:

- An error checking method used frequently in data communications and storage systems.
- A vital part of the ROM memory module search technique.
- Located at the end of all modules to check the validity of the entire module.
- An extremely reliable assurance that programs in memory are intact before execution.
- An effective backup for the error detection systems of disk drives and memory systems.



Note

The F$CRC description in Appendix D: OS-9 for 68K System Calls describes how F$CRC computes a module's CRC.

OS-9 computes a 24-bit CRC value 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 linkers automatically generate the module header and CRC values. If required, your program can use the F$CRC system call to compute a CRC value over any specified databytes.

OS-9 does not recognize a module with an incorrect CRC value. Therefore, you must update the CRC value of any ***patched*** or modified module for OS-9 to load the module from disk or find it in ROM. Use the OS-9 `fixmod` utility to update the CRC's of patched modules.

Note

The Atomic kernel's checking of the CRC can be disabled using the `M$SysConf` flags of the `init` module.

ROMed Memory Modules

When a system reset starts OS-9, the kernel searches for modules in ROM. It detects them by looking for the module header sync code ($4AFC). When the kernel detects this byte pattern, it checks the header parity to verify a correct header. If this test succeeds, the kernel obtains the module size from the header and computes a 24-bit CRC over the entire module. If the computed CRC is valid, the module is entered into the module directory.

OS-9 links to all of its component modules found during the search. It automatically includes in the system module directory all ROMed modules present in the system at startup. This allows you to create partially or completely ROM-based systems. It also includes any non-system modules found in ROM. This allows location of user-supplied software during the start-up process, and its entry into the module directory.

Module Header Definitions

The following table and Figure 1-4 list definitions of the standard set of fields in the module header.

Table 1-6 Module Header Fields

Name	Description
`M$ID`	**Sync Bytes ($4AFC)** These constant bytes identify the start of a module.
`M$SysRev`	**System Revision Identification** Identifies the format of a module.
`M$Size`	**Size of Module** The overall module size in bytes, including the header and CRC.
`M$Owner`	**Owner ID** 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. The name string can be located anywhere in the module and consists of a string of ASCII characters terminated by a null (zero) byte.

Table 1-6 Module Header Fields (continued)

Name	Description
M$Accs	**Access Permissions** Defines the permissible module access by its owner or other users. Module access permissions are divided into four sections: `reserved` (4 bits) `group` (4 bits) `public` (4 bits) `owner` (4 bits) Each of the non-reserved permission fields is defined as: `bit 3` reserved `bit 2` execute permission `bit 1` write permission `bit 0` read permission The total field is displayed as: `-----ewr-ewr-ewr`

Table 1-6 Module Header Fields (continued)

Name	Description
M$Type	**Module Type Code** Module type values are in the `oskdefs.d` file. They describe the module type code as:

	0	Not used (Wildcard value in system calls)
Prgm	1	Program module
Sbrtn	2	Subroutine module
Multi	3	Multi-module (reserved for future use)
Data	4	Data module
CSDData	5	Configuration status descriptor
	6-10	Reserved for future use
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

Table 1-6 Module Header Fields (continued)

Name	Description
M$Lang	**Language** You can find module language codes in the `oskdefs.d` file. They describe whether the module is executable and which language the run-time system requires for execution (if any): `0` — Unspecified language (Wildcard value in system calls) `Objct` `1` — 68000 machine language `ICode` `2` — Basic I-code `PCode` `3` — Pascal P-code `CCode` `4` — C I-code (reserved for future use) `CblCode` `5` — Cobol I-code `FrtnCode` `6` — Fortran `I-code` `7-15` — Reserved for future use `16-255` — User Definable **NOTE:** Not all combinations of module type codes and languages necessarily make sense.

Table 1-6 Module Header Fields (continued)

Name	Description
M$Attr	**Attributes** The bits are defined as follows: **Bit** — **The Module Is** 5 — A ***system state*** module. 6 — A sticky module. A sticky module is retained in memory when its link count becomes zero. The module is removed from memory when its link count becomes -1 or memory is required for another use. 7 — Re-entrant (sharable by multiple tasks).
M$Revs	**Module's Revision Level** If two modules with the same name and type are found in the memory search or loaded into memory, only the module with the highest revision level is kept. This enables easy substitution of modules for update or correction, especially ROMed modules.
M$Edit	**Edition** The software release level for maintenance. OS-9 does not use this field. Every time a program is revised (even for a small change), increase this number. We recommend you key internal documentation within the source program to this system.
M$Usage	**Comments** Reserved for offset to module usage comments (not currently used).

Table 1-6 Module Header Fields (continued)

Name	Description
`M$Symbol`	**Symbol Table Offset** Reserved for future use.
`M$Ident`	**Ident Code** (not currently used).
`M$HdExt`	**Module Header Extension**
`M$HdExtSz`	**Module Header Extension Size** The above two fields are used to identify the location of the additional header fields that are not part of the standard OS-9 Device Descriptor module headers. These fields are currently used only by the file manager using the DPIO source code system. The fields allow the DPIO I/O systems to specify an initialized data offset.
`M$Parity`	**Header Parity Check** The one's complement of the exclusive-OR of the previous header ***words***. OS-9 uses this for a quick check of the module's integrity.

Note

Offset refers to the location of a module field, relative to the starting address of the module. Resolve module offsets in assembly code by using the names shown here and linking the module with the relocatable library, `sys.l` or `usr.l`.

Table 1-7 Module Header Standard Fields

Offset	Name	Use
$00	M$ID	Sync Bytes ($4AFC)
$02	M$SysRev	Revision ID
$04	M$Size	Module Size
$08	M$Owner	Owner ID
$0C	M$Name	Module Name Offset †
$10	M$Accs	Access Permissions
$12	M$Type	Module Type
$13	M$Lang	Module Language
$14	M$Attr	Attributes
$15	M$Revs	Revision Level
$16	M$Edit	Edit Edition
$18	M$Usage	Usage Comments Offset †
$1C	M$Symbol	Symbol Table
$20	M$Ident	Ident Code
$22		Reserved
$28	M$HdEXT	Module Header Extension

Table 1-7 Module Header Standard Fields (continued)

Offset	Name	Use
$2C	M$HdExtSz	Module Header Extension Size
$2E	M$Parity	Header Parity Check
$30-up		Module Type Dependent
		Module Body
		CRC Check

† These fields are offset to strings

Additional Header Fields for Individual Modules

Program, trap handler, device driver, file manager, and system modules have additional standard header fields following the universal offsets. These additional fields are listed below and shown in **Figure 1-4**.

The *program module* is a common type of module (type: `Prgm`; language: `Objct`). A program module is executable as an independent process by the `F$Fork` or `F$Chain` system calls. The assembler and C compilers produce program modules, and most OS-9 commands are program modules. Program module headers have six fields in addition to the universal set.

For More Information

Chapter 4: Interprocess Communications describes trap handler modules. The ***OS-9 for 68K Processors Technical I/O Manual*** describes file manager modules and device driver modules.

Table 1-8 Additional Header Fields for Individual Modules

Name	Description
(Program, trap handler, device driver, file manager, and system module headers use the following two fields.)	
`M$Exec`	**Execution Offset** The offset to the program's starting address. In the case of a file manager or device driver, this is the offset to the module's entry table. For data modules, this is the offset to the module's sharable data.
`M$Excpt`	**Default User Trap Execution Entry Point** The relative address of a routine to execute if an uninitialized user trap is called.
(Program, trap handler, and device driver module headers use the following field.)	
`M$Mem`	**Memory Size** The required size of the program's data area (storage for program variables).
(Program and trap handler module headers use the following three fields.)	
`M$Stack`	**Stack Size** The minimum required size of the program's stack area.

Table 1-8 Additional Header Fields for Individual Modules (continued)

Name	Description
`M$IData`	**Initialized Data Offset** The offset to the initialization data area's starting address. This area contains values to copy to the program's data area. The linker places all constant values declared in `vsects` here. The first four-byte value is the offset from the beginning of the data area to which the initialized data is copied. The next four-byte value is the number of initialized data-bytes to follow.

Table 1-8 Additional Header Fields for Individual Modules (continued)

Name	Description
M$IRefs	**Initialized References Offset** The offset to a table of values to locate pointers in the data area. Initialized variables in the program's data area may contain values that are pointers to absolute addresses. Adjust code pointers by adding the absolute starting address of the object code area. Adjust the data pointers by adding the absolute starting address of the data area. The F$Fork system call does the effective address calculation at execution time using tables created in the module. 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 adjust. F$Fork makes the adjustment 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 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 LS word are given. This continues until a MS word of zero and a LS word of zero are found.

(Trap handler module headers use the following two fields.)

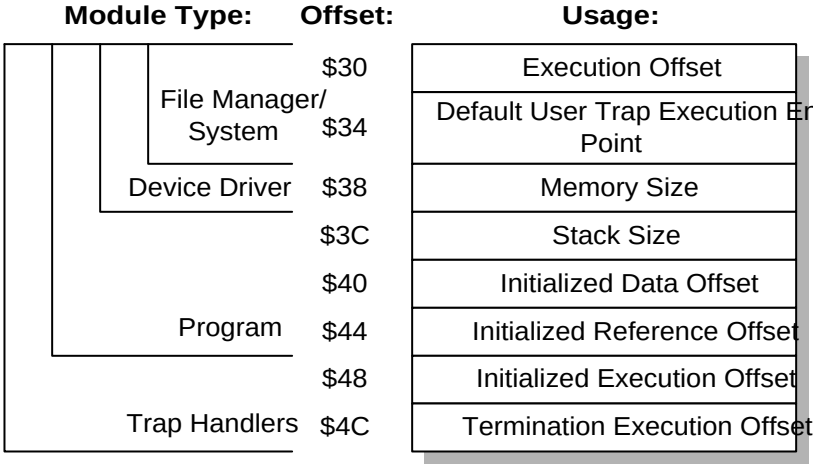
Table 1-8 Additional Header Fields for Individual Modules (continued)

Name	Description
`M$Init`	**Initialization Execution Offset** The offset to the trap initialization entry point.
`M$Term`	**Termination Execution Offset** The offset to the trap termination entry point. Microware reserves this offset for future use.

Note

Offset refers to the location of a module field, relative to the starting address of the module. Resolve module offsets in assembly code by using the names shown here and linking the module with the relocatable library: `sys.l` or `usr.l`.

Figure 1-4 Additional Header Fields for Individual Modules



Note

See Chapter 3: OS-9 Input/Output System for additional information on device descriptor's additional fields.

Chapter 2: The Kernel

This chapter outlines the responsibilities of the kernel. It explains user and system state processing, memory management, system utilization, process creation and scheduling, and exception and interrupt processing. It includes the following topics:

Responsibilities of the Kernel

The ***kernel*** is the nucleus of OS-9. It manages system resources, controls processing, and manages exception/interrupt processing. It is a ROMable, compact OS-9 module.

The kernel's primary responsibility is to process and coordinate system calls, or service requests. OS-9 has two general types of system calls:

- Calls performinng Input/Output, such as reads and writes.
- Calls performing system functions. System functions include memory management, system initialization, process creation and scheduling, and exception/interrupt processing.

When you make a system call, a user trap to the kernel occurs. The kernel determines what type of system call you want to perform. It directly executes the calls performing system functions, but does not execute I/O calls. The kernel provides the first level of processing for each I/O call, and then calls IOMan to complete the function by calling the appropriate file manager or device driver.

Kernel Types

There are two distinct classes of the kernel:

Development Kernel	is the full-featured version of the kernel designed to be usable in embedded and multi-user environments.
Atomic Kernel	is primarily designed to be used in embedded systems, although it can also be used in multi-user systems.

Kernel Differences

The main differences between the two versions of the kernel can be summarized as follows:

- ***User-state debugging*** (`F$DFork`, `F$DExec`, `F$DExit`) is not supported by the Atomic kernel. Debugging on Atomic systems is restricted to the capabilities of the system's ROM debugger.
- ***Multi-user protection mechanisms*** are not implemented in the Atomic kernel environment. These mechanisms include features such as:
  - Validation of user parameters. For example, validation of user buffer for data transfers.
  - External cache hardware is not supported. The Atomic kernel only supports on-chip caches (if present).
  - No MMU support for memory protection.
  - Validation of module/user ID permissions. Under an Atomic kernel (for example) any user can link to any module, regardless of the access permissions specified in the module.

These different kernel environments allow tailoring of the target system software to the requirements of the actual target application. Appendix D: OS-9 for 68K System Calls contains full details of the various system calls and in which environment they are available.

System Call Overview

For information about specific system calls, refer to Appendix D: OS-9 for 68K System Calls.

User-State and System-State

To understand OS-9's system calls, you should be familiar with the two distinct OS-9 environments in which you can execute object code:

User-State	The normal program environment in which processes execute. Generally, user-state processes do not deal directly with the specific hardware configuration of the system.
System-State	The environment in which OS-9 system calls and interrupt service routines execute. On 68000-family processors, this is synonymous with supervisor state. System-state routines often deal with physical hardware present on a system.

Functions executing in system state have distinct advantages over those running in user state, including the following:

- A system-state routine can access all of the processor's capabilities. For example, on memory protected systems, a system-state routine may access any memory location in the system. It may mask interrupts, alter OS-9 internal data structures, or take direct control of hardware interrupt vectors.
- Some OS-9 system calls are only accessible from system state.

System-state characteristics make it the only way to provide certain types of programming functions. For example, it is almost impossible to provide direct I/O to a physical device from user state. Not all programs, however, should run in system state. Reasons to use user-state processing rather than system-state processing include:

- Memory protection prevents user-state routines from accidentally damaging data structures they do not own.
- A user-state process can be aborted. If a system-state routine loses control, the entire system usually crashes.
- System-state routines are far more difficult and dangerous to debug than user-state routines. You can use the user-state debugger to find most user-state problems. Generally, system-state problems are much more difficult to find.
- User-state programs are essentially isolated from physical hardware. Because they are not concerned with I/O details, they are easier to write and port.

Installing System-State Routines

System-state routines have direct access to all system hardware and can take over the entire machine, crashing or hanging up the system. To help prevent this, OS-9 limits the methods of creating routines that operate in system state.

There are four ways to provide system-state routines:

1. Install an 0S9P2 module in the system bootstrap file or in ROM. During cold start, the OS-9 kernel links to this module, and if found, calls its execution entry point. The most likely thing for such a module to do is install new system call codes. The drawback to this method is the 0S9P2 module must be in ROM or in the bootfile when the system is bootstrapped.
2. Use the I/O system as an entry into system state. File managers and device drivers always execute in system state. The most obvious reason to write system-state routines is to provide support for new hardware devices. You can write a dummy device driver and use the `I$GetStt` or `I$SetStt` routines to provide a gateway to the driver.
3. Write a trap handler module that executes in system state. In many cases, it is practical to debug most of the trap handler routines in user state, and then convert the trap module to system state. To make a trap handler execute in system state, you must set the

supervisor state bit in the module attribute byte and create the module as super user. When the user trap executes, it is in system state.

4. A program executes in system state if the supervisor state bit in the module's attribute word is set and the module is owned by the super user. This can be useful in rare instances.

Note

For routines of limited use that are dynamically loaded and unlinked, writing a trap handler module may be the most convenient method.

Note

I/O-related system-state routines may not be time-sliced. The kernel disables system state time-slicing for a process when it dispatches to IOMan. It is the responsibility of the file manager and/or device driver (rarely) to allow system state time-slicing for the process while the I/O request is executed.

Kernel System Call Processing

The kernel processes all OS-9 system calls (service requests). The system-wide relocatable library files, `sys.l` and `usr.l`, define symbolic names for all system calls. 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 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. There are two general types of system calls:

- System function calls (calls that do not perform I/O).
- I/O calls.

System Function Calls

There are two types of system function calls, user-state and system-state:

User-State	These requests perform memory management, multitasking, and other functions for user programs. They are mainly processed by the kernel.
System-State	Only system software in system state can use these calls, and they usually operate on internal OS-9 data structures. To preserve OS-9's modularity, these requests are system calls rather than subroutines. User-state

programs cannot access them, but system modules such as device drivers may use them.

The symbolic name of each system function call begins with F$. For example, the system call to link a module is F$Link.

Memory requested in system state is ***not*** recorded in the process descriptor memory list. Therefore, you must ensure the memory is returned to the system before the process terminates.

Note

In general, system-state routines may use any of the user-state system calls. However, you must avoid making system calls at inappropriate times. For example, avoid I/O calls, timed sleep requests, and other calls that can be particularly time consuming (such as F$CRC) in an interrupt service routine.

WARNING

Avoid the F$TLink and F$Icpt system calls in system-state routines. Certain portions of the C library may be inappropriate for use in system state.

I/O Calls

I/O calls perform various I/O functions. IOMan, the file manager, device driver, and kernel process I/O calls for a particular device. The symbolic names for this category of calls begin with I$. For example, the read service request is I$Read.

You may use any I/O system call in a system-state routine, with one slight difference than when executed in user-state. All path numbers used in system state are system path numbers. Each process descriptor has a path table that converts process local path numbers into system path numbers. The system itself has a global path number table to convert system path numbers into actual addresses of path descriptors. You must make system-state I/O system calls using system path numbers.

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

When a user-state process exits with open I/O paths, the `F$Exit` routine automatically closes the paths. This is possible because OS-9 keeps track of the open paths in the process' path table. In system state, the `I$Open` and `I$Create` system calls return a system path number that OS-9 does not record in the process path table or anywhere else. Therefore, the system-state routine that opens any I/O paths must ensure the paths are eventually closed, even if the underlying process is abnormally terminated.

Memory Management

To load any object (such as a program or constant table) into memory, the object ***must*** have the standard OS-9 memory module format as described in Chapter 1: System Overview. This allows OS-9 to maintain a ***module directory*** to keep track of modules in memory. The module directory contains the name, address, and other related information about each module in memory.

OS-9 adds the module to the module directory when it is loaded into memory. Each directory entry contains a ***link count***. The link count is the number of processes using the module.

When a process links to a module in memory, the module's link count increments by one. When a process unlinks from a module, the module's link count decrements by one. When a module's link count becomes `0`, its memory is de-allocated and it is removed from the module directory, unless the module is ***sticky***.

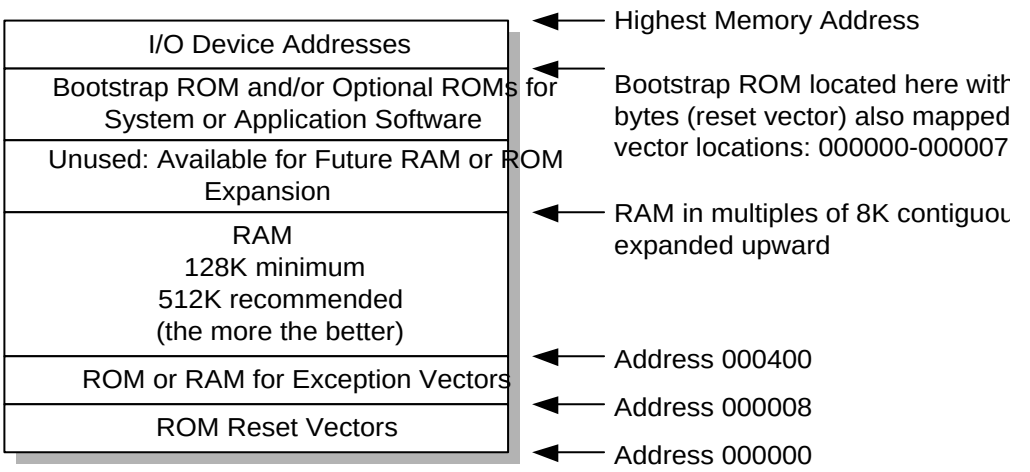
A ***sticky module*** is not removed from memory until its link count becomes -1 or memory is required for another use. A module is sticky if the sixth bit of the module header's attribute field (`M$Attr`) is set.

OS-9 Memory Map

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

A map of a typical OS-9 memory space is shown in **Figure 2-1**. Unless otherwise noted, the sections shown need not be located at specific addresses. However, Microware recommends you keep each section in contiguous reserved blocks, arranged in an order that facilitates future expansion. Whenever possible, it is best to have physically contiguous RAM.

Figure 2-1 Typical OS-9 Memory Map



Note

For the 68020, 68030, 68040, and CPU32 family of CPUs, you can set the Vector Base Register (VBR) anywhere in the system. Thus, for these types of systems, there is no requirement that RAM or ROM be at address 0.

System Memory Allocation

During the OS-9 start-up sequence, an automatic search function in the kernel and the boot ROM finds blocks of RAM and ROM. OS-9 reserves some RAM for its own data structures. ROM blocks are searched for valid OS-9 ROM modules.

OS-9 requires a variable amount of memory. 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, IOMan, file managers, device drivers, device descriptors, and tick driver) occupies about 50K to 64K bytes of memory. On disk-based systems, these modules are normally bootstrap-loaded into RAM. OS-9 does not dynamically load overlays or swap system code; therefore, no additional RAM is required for system code.

You can place OS-9 in ROM for non-disk systems. The typical operating system object code for ROM-based, non-disk systems occupies about 30K to 40K bytes.

OS-9 uses a minimum of 8K RAM for internal use. The system global memory area is usually located at the lowest RAM addressed. It contains an exception jump table, the debugger/boot variables, and a system global area. Variables in the system global area are symbolically defined in the `sys.l` library and the variable names begin with `D_`. The Reset SSP vector points to the system global area.

WARNING

User programs should ***never*** directly access system global variables because of issues such as portability and (depending on hardware) memory protection. System calls are provided to allow user programs to read the information in this area.

OS-9 maintains dynamic-sized data structures (such as I/O buffers, path descriptors, and process descriptors) that are allocated from the general RAM area when needed. The System Global Memory area keeps pointers to the addresses of these data structures. A typical small system uses approximately 32K of RAM. The total depends on elements such as the number of active devices, the memory, and the number of active processes. The `sys.l` library source files include the exact sizes of all the system's data structure elements.

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:

- Modules are loaded into RAM.
- New processes are created.
- Processes request additional RAM.
- OS-9 requires 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 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 required for user memory depends largely on the application software to be run. Microware suggests you have a system minimum of 128K plus an additional 64K per user available. Alternatively, a small ROM-based control system might only need 64K of memory.

Memory Allocators

The kernel offers two versions of the memory allocator:

- Buddy Allocator
- Standard Allocator

The allocator used is determined by the version of the kernel you select.

The Buddy Allocator is designed to give more deterministic operation in a real-time environment than the Standard Allocator, at the expense of memory efficiency.

The majority of systems use the Standard Allocator. This is the memory allocator used by the kernel since the original release of OS-9. This allocator can allocate memory to a resolution of 16-bytes. For example, a request for 1025 bytes rounds up to 1040 bytes.

The Buddy Allocator uses a ***binary-buddy*** algorithm which maintains free memory in block sizes that are binary multiples. Under the Buddy Allocator, memory requests are rounded up to the next binary power of the request size. For example, a request for 1025 bytes rounds up to 2048 bytes. The binary-buddy technique results in faster memory allocation for the system, and thus overall system determinism is improved. Because the Buddy Allocator is less efficient for system memory usage, this allocator is typically used in embedded (the Atomic kernel) applications where real-time performance is critical.

When you use the buddy allocator, the system memory lists (either in ROM or in the `Init` module's colored memory lists) must have a start address on the same boundary as the size of the block.

Memory Fragmentation

Once a program is loaded, it must remain at the address where it was originally loaded. Although position-independent programs can be initially placed at any address where free memory is available, program modules cannot be relocated dynamically after they are loaded. This can lead to ***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 colored memory request is made, this may not be true. If a number of program modules are loaded, and subsequently one or more non-contiguous modules are unlinked, several fragments of free memory space exist. The total free memory space may be quite large. However, because it is scattered, not enough space exists in a single block to load a particular program module.

For More Information

For more information on colored memory, refer to the next section.

You can avoid memory fragmentation by loading modules at system startup. This places the modules in contiguous memory space. Or, you can initialize each standard device when booting the system. This allows the devices to allocate memory from higher RAM than would be available if the devices were initialized after booting.

For More Information

`mfree` displays free system RAM. Refer to the ***Utilities Reference*** for more information about `mfree`.

If serious memory fragmentation does occur, the system administrator can kill processes and unlink modules in ascending order of importance until there is sufficient contiguous memory to proceed. Use the `mfree` utility to determine the number and size of free memory blocks.

Colored Memory

OS-9 colored memory allows a system to recognize different memory types and reserve areas for special purposes. For example, you could design a part of a system's RAM to store video images and battery back up another part. The kernel allows isolation and specific access of areas of RAM like these. You can request specific memory types or ***colors*** when:

- Allocating memory buffers.
- Creating modules in memory.
- Loading modules into memory.

If a specific type of memory is not available, the kernel returns error #237, `E$NoRAM`.

Colored memory lists are not essential on systems with RAM consisting of one homogeneous type, although they can improve system performance on some systems and allow greater flexibility in configuring memory search areas. The default memory allocation requests are still appropriate for most homogeneous systems and for applications that do not require one memory type over another. Colored memory lists are required for the `F$Trans` system call to perform address translation.

Colored Memory Definition List

The kernel must have a description of the CPU's address space to make use of the colored memory routines. You can establish colored memory by including a colored memory definition list (`MemList`) in the `systype.d` file, which then becomes part of the `Init` module. The list describes each memory region's characteristics. The kernel searches each region in the list for RAM during system startup.

A colored memory definition list contains the following information:

- Memory color (type)
- Memory priority

- Memory access permissions
- Local bus address
- Block size the kernel's coldstart routine uses to search the area for RAM or ROM
- External bus translation address (for DMA and dual-ported RAM)
- Optional name

The memory list may contain as many regions as needed. If no list is specified, the kernel automatically creates one region describing the memory found by the bootstrap ROM.

`MemList` is a series of `MemType` macros defined in `systype.d` and used by `init.a`. Each line in the `MemList` must contain all the following parameters, in order:

```
type, priority, attributes, blksiz, addr begin,
addr end, name, DMA-offset
```

The colored memory list must end with a longword of zero. The following describes the `MemList` parameters:

Table 2-1 MemList Parameters

Parameter	Size	Definition
Memory Type	word	Type of memory. Three memory types are currently defined in `memory.h`: `SYSRAM` `0x01` System RAM memory `VIDEO1` `0x80` Video memory for plane A `VIDEO2` `0x81` Video memory for plane B
Priority	word	Priority of memory (0-255). High priority memory is allocated first. If the block priority is `0`, the block can only be allocated by a request for the specific color (type) of the block.

Table 2-1 MemList Parameters (continued)

Parameter	Size	Definition
Access Permissions	word	Memory type access bit definitions: **Bit Name — Description** 0 B_USER — User processes can allocate this memory. **NOTE:** This bit is ignored if the B_ROM bit is set. 1 B_PARITY — Parity memory; the kernel initializes this memory during startup. **NOTE:** Only B_USER memory may be initialized. 2 B_ROM — ROM; the kernel searches this memory for modules during startup. **NOTE:** Processes cannot allocate B_ROM memory, as the B_USER and B_PARITY bits are ignored if B_ROM is set. 3 B_NVRAM — NVRAM memory is searched for modules. 4 B_SHARE — SHARED memory is supported by allocating the system's data structure for the shared memory block out of the block itself.
Search Block Size	word	The kernel checks every (search block size)<<4 to see if RAM/ROM exists.

Table 2-1 MemList Parameters (continued)

Parameter	Size	Definition
Low Memory Limit	long	Beginning address of the block, as referenced by the CPU.
High Memory Limit	long	End address of the block (+1), as referenced by the CPU.
Description String Offset	word	Offset of a user-defined string describing the type of memory block.
Address Translation Adjustment	long	The external bus address of the beginning of the block. If `0`, this field does not apply. Refer to `F$Trans` for more information.

The following is an example system memory map:

```
CPU Address     Bus Address     Memory Size     Physical Location
$00000000       $00200000       $200000         on-board cpu ram
$00600000       $00600000       $200000         VMEbus ram
```

A corresponding `MemList` table might appear as follows:

```
* memory list definitions for init module (user adjustable)
 align
* MemType type, prior, attributes, blksiz, addr limits, name, DMA-offset
MemList
* on-board ram covered by "rom memory list:"
*  - this memory block is known to the "rom's memory list," thus it was
*    "parity initialized" by the rom code.
*  - the cpu's local base address of the block is at $00000000.
*  - the bus base address of the block is at $200000.
*  - this ram is fastest access for the cpu, so it has the highest priority.
*
 MemType SYSRAM,255,B_USER,4096,0,$200000,OnBoard,$200000

* off-board expansion ram
*  - this memory block is not known to the "rom's memory list,"
*    thus it needs "parity initialization" by the kernel.
*  - as the block is accessed over the bus, the base address of the block
*    is the same for cpu and dma accesses.
*  - this ram is slower access than on-board ram, therefore it
*    has a lower priority than the on-board ram.
*
 MemType SYSRAM,250,B_USER+B_PARITY,4096,$600000,$800000,OffBoard,0
```

```
 dc.l 0 end of list

OnBoard dc.b "fast on-board RAM",0
OffBoard dc.b "VMEbus memory",0
```

Colored memory definitions are not essential for homogenous memory systems. However, colored memory definitions in this type of system can improve system performance and simplify memory list reconfiguration.

In a homogeneous memory system, the kernel allocates memory from the top of available RAM when requests are made by F$SRqMem (for example, when loading modules). If the system has RAM on-board the CPU and off-board in external memory boards, the modules tend to be loaded into the off-board RAM, because OS-9 always uses high memory first. On-board RAM is not used for a F$SRqMem call until the off-board memory is unable to accommodate the request.

Programs running in off-board memory execute slower than those running in on-board memory, due to bus access arbitration. Also, external bus activity increases. This may impact the performance of other bus masters in the system.

The colored memory lists can be used to reverse this tendency in the kernel, so a CPU does not use off-board memory until all of its on-board memory is used. This results in faster program execution and less saturation of the system's external bus. Do this by making the priority of the on-board memory higher than off-board memory, as shown in the example lists on the preceding page.

In a homogeneous memory system, the memory search areas are defined in the ROM's memory list. If you do not use colored memory, you must make new ROMs from source code (usually impossible for end-users) or from a patched version of the original ROMs (usually difficult for end-users) to make changes to the memory search areas.

The colored memory lists simplify changes by configuring the search areas as follows:

- The ROM's memory list describes only the on-board memory.
- The colored memory lists in `systype.d` define the on-board memory and any external bus memory search areas in the `Init` module only.

The use of colored memory in a homogeneous memory system allows you to easily reconfigure the external bus search areas by adjusting the lists in `systype.d` and making a new `Init` module. The ROM does not require patching.

System Memory Cache Lists

OS-9 supports the ability to precisely define the caching modes used for regions of memory in the system. Precise definition of these modes for particular regions allows the user to configure the system for optimal performance and/or system functionality.

Many systems simply desire maximum performance, but there are many cases where regions of memory must be declared non-cachable so cache coherency problems do not result when processes directly reference I/O devices and memory shared with other processors.

When the SSM module is installed in the system, it provides a default cache mode of write-through for user-state accesses. This default mode can be over-ridden for specific regions by creating `CacheList` entries in the `Init` module.

A `CacheList` entry contains the following information:

- Memory block start address.
- Memory block end address.
- Cache mode for memory block.

The `CacheList` may contain as many regions as needed. If no list is specified, then the default cache mode for user-state is write-through.

`CacheList` is a series of `CacheType` macros defined in `systype.d` and used by `init.a`. Each line in the `CacheList` must contain the following parameters, in order:

```
block start, block end, cache mode
```

The cache list entries must end with a longword of 0xffffffff (-1). The following describes the `CacheList` parameters:

Table 2-2 CacheList Parameters

Parameter	Size	Definition
Block Start	long	Start address of memory region.
Block End	long	End address (+1) of memory region.
Cache Mode	word	Cache mode (MMU specific) for region.

The following is an example system memory map:

Table 2-3 Example System Memory Map

CPU Address	Size	Memory Size, Usage, and Cache Mode
`$00000000`	`$02000000`	32Meg on-board memory, cache enabled, copy-back mode.This memory can be ***snooped*** by the on-board master, so copy-back is viable.
`$02000000`	`$7e000000`	Off-board memory. This region cannot be snooped by the on-board master, but as it is being used for on-board master usage only, then copy-back caching is viable.

Table 2-3 Example System Memory Map (continued)

CPU Address	Size	Memory Size, Usage, and Cache Mode
$800000000	$02000000	32Meg reserved for shared memory with other processors or a memory-mapped video (for example). This region should be cache-inhibited (shared case) or for performance (video case).
$e0000000	$20000000	System I/O regions (both on-board peripherals and I/O expansion space). This region must be cache inhibited and serialized access due to the nature of I/O devices.

A corresponding CacheList Table might appear as follows:

```
* CacheList entries for init module (user adjustable)
*
* define cache-mode overrides for system
*
 CPUALIGN
CacheList
 CacheType 0,$02000000,CopyBack on-board area is copy-back
 CacheType $02000000,$80000000,CopyBack off-board area is copy-back
 CacheType $80000000,$82000000,CINotSer shared area is cache-inhibited
 CacheType $e0000000,$ffffffff,CISer I/O area us cache-inhibited, serialized
 dc.l -1 terminate list
```

Note

There is no direct relationship between the kernel's colored memory lists and the CacheList entries. Each list is used for its own purpose. For example, it is allowed to have two cache modes described for separate regions falling within a single colored memory list entry.

System Initialization

After a hardware reset, the bootstrap ROM executes the kernel (located in ROM or loaded from disk, depending on the system involved). The kernel initializes the system, which includes locating ROM modules and running the system startup task (usually `Sysgo`).

Init: The Configuration Module

`Init` is a non-executable module of type `Systm` (code $0C) which contains a table of system startup parameters. During startup, `Init` specifies initial table sizes and system device names, but it is always available to determine system limits. It must be in memory when the kernel executes and usually resides in the `OS9Boot` file or in ROM.

The `Init` module begins with a standard module header (the standard module header is covered in Chapter 1: System Overview) and the additional fields shown in **Table 2-4** and in **Figure 2-2**.

Note

Refer to Appendix A: Example Code for an example program listing of the `Init` module. Offset names are defined in the relocatable library `sys.l`.

Table 2-4 Init Module Values

Offset	Name	Description
$30	Reserved	Reserved for future use.
$34	M$PollSz	***Number of Entries in the IRQ Polling Table*** One entry is required for each interrupt generating device control register. Atomic kernel default is 16. Development kernel default is 32.
$36	M$DevCnt	***Device Table Size*** The number of entries in the system device table. One entry is required for each device in the system. Atomic IOMan default is 8. Development IOMan default is 32.
$38	M$Procs.	***Initial Process Table Size*** Indicates the initial number of active processes allowed in the system. For Atomic OS-9, this table is fixed; for the development kernel, it automatically expands as needed. Atomic kernel default is 32. Development kernel default is 64.
$3A	M$Paths	***Initial Path Table Size*** The initial number of open paths in the system. For Atomic OS-9, this table is fixed; for the development kernel, it automatically expands as needed. Atomic IOMan default is 32. Development IOMan default is 64.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$3C	M$SParam	**Offset to Parameter String for Startup Module** The offset to the parameter string (if any) to pass to the first executable module. An offset of 0 indicates no parameter string is required. The parameter string itself is located elsewhere, usually near the end of the Init module.
$3E	M$SysGo	***First Executable Module Name Offset*** The offset to the name string of the first executable module; usually SysGo or shell.
$40	M$SysDev	***Default Directory Name Offset*** The offset to the initial default directory name string; usually /d0 or /h0. The kernel does a chd and chx to this device before forking the initial device. If the system does not use disks, this offset must be 0.
$42	M$Consol	***Initial I/O Pathlist Name Offset*** This offset usually points to the /TERM string. This pathlist is opened as the standard I/O path for the initial process. It is generally used to set up the initial I/O paths to and from a terminal. This offset should contain 0 if no console device is in use.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$44	M$Extens	***Customization Module Name Offset*** The offset to a name string of a list of customization modules (if any). A customization module is intended to complement or change OS-9's existing standard system calls. OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system state if found. Modules listed in the name string are separated by spaces. The default name string to search for is OS9P2. If there are no customization modules, set this value to 0. **Note:** A customization module may only alter the d0, d1, and ccr registers. **More Information:** Refer to the following section for more information on customization modules.
$46	M$Clock	***Clock Module Name Offset*** If there is no clock module name string, set this value to 0.
$48	M$Slice	***Time slice*** The number of clock ticks per time slice. The number of clock ticks per time slice defaults to 2.
$4A	Reserved	Reserved for future use.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$4C	M$Site	***Installation Site Code Offset*** This value is usually set to 0. OS-9 does not currently use this field.
$50	M$Instal	***Offset to Installation Name***
$52	M$CPUTyp	***CPU Type*** CPU type: 68000, 68008, 68010, 68020, 68030, 68040, 68070, or 683XX. The default is 68000.
$56	M$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 3.0, edition 1 would be 1301.
$5A	M$OS9Rev	***Revision Offset*** The offset to the OS-9 level/revision string.
$5C	M$SysPri	***Priority*** The system priority at which the first module (usually SysGo or shell) is executed. This is generally the base priority at which all processes start. The default is 128.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$5E	M$MinPty	***Minimum Priority*** The initial system minimum executable priority. The default is 0.
$60	M$MaxAge	***Maximum Age*** The initial system maximum natural age. The default is 0.
$62	M$MDirSz	***Module Directory Size*** The initial module count for the system. For the Atomic kernel this table is fixed; for the Development kernel, it automatically expands as needed. Atomic and Development kernels' defaults are 64.
$64	Reserved	Reserved for future use.
$66	M$Events	***Number of Entries in the Events Table*** The initial number of entries allowed in the events table. For the Atomic kernel this table is fixed; for the Development kernel, it automatically expands as needed. Atomic kernel default is 16. Development kernel default is 32. **More Information:** Refer to F$Event for a discussion of using events.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$68	M$Compat	***Revision Compatibility*** This byte is used for revision compatibility. The default is 0. The following bits are currently defined: **Bit Set Bit To:** 0 Save all registers for IRQ routines. If you have OS-9 for 68K version 3.0 or greater, this flag is ignored. 1 Prevent the kernel from using stop instructions. 2 Ignore ***sticky*** bit in module headers. 3 Disable cache burst operation (68030 systems). 4 Patternize memory when allocated or de-allocated. 5 Prevent kernel cold-start from starting system clock. 6 Kernel ignores spurious IRQs. 7 Only the process creating an alarm can delete it.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$69	M$Compa 2	***Compatibility Bit #2*** This byte is used for revision compatibility. The following bits are currently defined:

Bit		Function
0	0	External instruction cache is *not* snoopy.*
	1	External instruction cache is snoopy or absent.
1	0	External data cache is *not* snoopy.
	1	External data cache is snoopy or absent.
2	0	On-chip instruction cache is *not* snoopy.
	1	On-chip instruction cache is snoopy or absent.
3	0	On-chip data cache is *not* snoopy.
	1	On-chip data cache is snoopy or absent.
4	0	68349: Cache/SRAM Bank 0 is SRAM.
	1	68349: Cache/SRAM Bank 0 is Cache.
5	0	68349: Cache/SRAM Bank 1 is SRAM.
	1	68349: Cache/SRAM Bank 1 is Cache.
6	0	68349: Cache/SRAM Bank 2 is SRAM.
	1	68349: Cache/SRAM Bank 2 is Cache.
7	0	68349: Cache/SRAM Bank 3 is SRAM.
	1	68349: Cache/SRAM Bank 3 is Cache.

* snoopy = cache that maintains its integrity without software intervention.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$6A	M$MemList	***Colored Memory List Offset*** The colored memory list contains an entry for each type of memory in the system. The list is terminated by a long word of 0. If this field contains a 0, colored memory is not used in this system. **More Information:** For a complete discussion on colored memory, refer to **Colored Memory** earlier in this chapter.
$6C	M$IRQStk	***Size of Kernel's IRQ Stack*** This field contains the size (in longwords) of the kernel's IRQ stack. The value must be 0 or between 256 and $ffff. If the value is zero, the kernel uses a small default IRQ stack. A larger IRQ stack is recommended. The default value is 256 longwords.
$6E	M$ColdTrys	***Retry Counter*** The retry counter if the kernel's initial chd to the system device fails. The default value is 0.
$70	Reserved	
$72	Reserved	

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$74	M$CacheList	***Cache List Offset*** The cache list entries describe alternate cache modes for user-state accesses to memory regions. The list is terminated by a long word of -1. If this field is 0, the cache lists are not used in the system. **More Information:** For a complete discussion on cache lists, refer to **System Memory Cache Lists** earlier in this chapter.
$76	M$IOMan	***I/O Manager Module Name Offset*** The offset to a name string of a list of I/O manager modules (if any). OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system-state if found. Modules listed in the name string are separated by spaces. The default name to search for is IOMan. If there are no I/O modules, set this to 0. **Note:** The I/O modules may only alter the d0, d1, and ccr registers. **More Information:** Refer to **Customization Modules** later in this chapter for information about customization modules.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$78	M$PreIO	***Pre-I/O Module Name Offset*** The offset to a name string of a list of Pre-I/O modules (if any). OS-9 searches for these modules during startup. Typically, these modules are found in the bootfile. They are executed in system-state if found. Modules listed in the name string are separated by spaces. The default name to search for is `PreIO`. If there are no Pre-I/O modules, set this to `0`. **Note:** The I/O modules may only alter the `d0`, `d1`, and `ccr` registers. **More Information:** Refer to **Customization Modules** later in this chapter for information about customization modules.

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$7a	M$SysConf	***System Configuration Flags*** This word field is used for system configuration control. The following bits are currently defined: **Bit** **Function** 0 0 System tables are expanded as needed. 1 System table overflow results in an error. The default values in the table are set in the Init module. **Note:** System table expansion only applies to the Development kernel. For the Atomic kernel, table sizes are fixed from the Init module values. 1 Reserved 2 0 CRC checking performed by F$VModul. 1 CRC checking disabled for F$VModul. **Note:** CRC check disabling applies only to the Atomic kernel and only for checks made after cold start. 3 0 System-state time-slicing enabled. 1 System-state time-slicing disabled. 4 0 SSM builds user-state protection tables on a per-process basis 1 SSM builds one user-state page table (to allow access to all known memory) at cold-start. **Note:** This option only applies to the development kernel. The atomic kernel case always builds a single user-state page table. 5 - 15 Reserved

Table 2-4 Init Module Values (continued)

Offset	Name	Description
$7c	Reserved	
$7e	M$PrcDescStack	***Size of Process Descriptor's Stack*** This field determines the stack area size in a process descriptor. This stack is used by the process when it is performing system calls (for example, I/O operations). Systems with file managers/drivers using the stack heavily (for example, drivers written in C) may need to increase this value. The default is 1500 bytes.

Note

Note the following:

- Throughout this chapter, the system directories referred to are the defaults found in the `Init` module, unless otherwise specified.
- 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`.

Table 2-5 Additional Fields for the Init Module

Offset	Name	Description
$30	Reserved	Currently reserved for future use.
$34	M$PollSz	Number of IRQ polling table entries.
$36	M$DevCnt	Device table size.
$38	M$Procs	Initial process table size.
$3A	M$Paths	Initial path table size.
$3C	M$SParam	Parameter string for startup module (usually Sysgo).
$3E	M$SysGo	Offset to name string of first executable module.
$40	M$SysDev	Offset to the initial default directory name string.
$42	M$Consol	Offset to the initial I/O pathlist string.
$44	M$Extens	Offset to a name string of customization modules.
$46	M$Clock	Offset to the clock module name string.
$48	M$Slice	Number of clock ticks per timeslice.
$4A	Reserved	Currently reserved for future use.
$4C	M$Site	Offset to the installation site code.

Table 2-5 Additional Fields for the Init Module (continued)

Offset	Name	Description
$50	M$Instal	Offset to the installation name string.
$52	M$CPUTyp	CPU type.
$56	M$OS9Lvl	Level, version, and edition number of the operating system.
$5A	M$OS9Rev	Offset to the OS-9 level/revision string.
$5C	M$SysPri	Initial system priority.
$5E	M$MinPty	Initial system minimum executable priority.
$60	M$MaxAge	Initial system maximum natural age.
$62	M$MDirSz	Module directory size
$64	Reserved	Currently reserved for future use.
$66	M$Events	Initial number of entries allowed in the events table.
$68	M$Compat	Compatibility flag one. Byte is used for revision compatibility.
$69	M$Compat2	Compatibility flag two. Byte is used for cache control.
$6A	M$MemList	Offset to the memory segment list.
$6C	M$IRQStk	Size of the kernel's IRQ stack.

Table 2-5 Additional Fields for the Init Module (continued)

Offset	Name	Description
$6E	M$ColdTrys	Retry counter if the kernel's initial chd fails.
$70	Reserved	Currently reserved for future use.
$72	Reserved	Currently reserved for future use.
$74	M$CacheList	Offset to the cache list.
$76	M$IOMan	Offset to the I/O manager name.
$78	M$PreIO	Offset to the Pre-I/O module list.
$7a	M$SysConf	System configuration. This word is used to tailor specific system options.
$7c	Reserved	Currently reserved for future use.
$7e	M$PrcDescStack	Size of process descriptor stack.

Initial System Process

The first process executed by OS-9 is the process named in the `Init` module. When the kernel has finished its system initialization, the process is forked and any parameters specified in the `Init` module is passed to that process.

How the system is initially started is dependent upon the requirements of the system; it can be one of the standard `Sysgo` modules or it can be any custom procedure you desire.

The standard modules for starting the system are called `Sysgo`. If you are using a disk-based system then the default `Sysgo` module can be used to read the disk file called `startup`. If you have a non-disk based system, another version of `Sysgo` is available (`sysgo_nodisk`) to bring up a shell without using any disk for I/O.

The `Sysgo` for disk operation operates as follows:

`Sysgo` is the first user process started after the system startup sequence. Its standard I/O is on the system console device.

`Sysgo` usually executes as follows:

Step 1. Change to the `CMDS` execution directory on the system device.

Step 2. Execute the `startup` file (as a script) from the root of the system device.

Step 3. Fork a shell on the system console.

Step 4. Wait for that shell to terminate and then forks it again. Therefore, there is always a shell running on the system console, unless `Sysgo` dies.

For More Information

Appendix A: Example Code contains an example source listing of the Sysgo module.

You may eliminate Sysgo by specifying shell as the initial module and specifying a parameter string similar to:

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

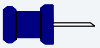
The sysgo for non-disk operation operates as follows:

Step 1. Fork a shell with standard I/O paths set to the system's default console device.

Step 2. Wait for that shell to terminate, and then forks Shell again. Thus, there is always Shell running on the system console unless Sysgo dies.

Customization Modules

Customization modules are additional modules you can execute at boot time to enhance OS-9's capabilities. They provide a convenient way to install a new system call code or collection of system call codes, such as a system security module. The kernel calls the modules at boot time if their names are specified in the extension list of the `Init` module and the kernel can locate them.



Note

Customization modules may only modify the `d0`, `d1`, and `ccr` registers.

In the `Init` module, the `M$Extens` offset points to a list of module names. By default, the module name in the list is `OS9P2`. If the modules are found during cold-start, they are called. If an error is returned, the system stops. The most common modules used are listed here:

Syscache

The `syscache` module allows the system to enable and control any hardware caches present. The default `syscache` module supplied by Microware controls the on-chip cache(s) for the processor being used. You can customize this module to use any external (off-chip) cache hardware the system may have. The `syscache` module installs the `F$CCtl` system call routines. If you do not install the `syscache` module, no system caching takes place.

Standard `syscache` modules (to support the on-chip capabilities of the processor) are provided for the 68020, 68030, 68040, and 68349.

Note

External hardware caches are only supported by the Development kernel. On-chip caching is supported by both the Atomic and Development kernels.

SSM

The system security module (SSM) allows operation of the memory management unit (MMU) for the processor in use.

For the Development kernel, MMU operation under OS-9 provides the basic user-state protection mechanisms for the system so user-state processes only access memory they are allowed to access. For the 68040 processors, the SSM module (in conjunction with the `CacheList` entries of the `Init` module) also allows fine-tuning of the system memory's cache attributes, so cache modes other than the default write-through mode are possible (for example copy-back and non-cacheable regions).

For the Atomic kernel, user-state protection is not implemented. Thus, SSM modules that do not provide cache support (all SSM modules except the 68040 SSM) should not be used. In an Atomic kernel environment, the 68040 SSM module provides just the cache mode support described above.

The standard SSM modules provided are:

SSM451	Supports the MC68451 MMU, for 68010 systems. This SSM module only provides protection functions.
SSM851	Supports the MC68851 PMMU and MC68030 MMU. Used on MC68020 systems and MC68030 systems. This SSM module only provides protection functions.

SSM040	Supports the MC68030 MMU. Used on MC68040 systems. This SSM module provides protection functions (if running in a Development kernel environment) as well as cache mode support.

FPU/FPSP

The FPU/FPSP modules provide floating point emulation and support functions for the system.

FPSP is used on MC68040 systems possessing an on-chip FPU and is used to provide software emulation of MC68881/2 instructions that are not implemented on the MC68040.

FPU is used on all other processors when the system does not have a hardware floating point unit (MC68EC040, MC68030 without MC68882 FPCP). This module provides emulation support for the MC68882 floating point unit.

Including a Customization Module

To include a customization module in the system, you can either burn the module into ROM or complete the following steps:

Step 1. Assemble/link the customization code to create an OS-9 system module.

Note

`os9p2` is the name of the default customization module.

Step 2. Create a new `Init` module:

- Edit the `CONFIG` macro in the `systype.d` file for your CPU board. The name of the new module must appear in the `Init` module extension list. For example, if the name of the new module is `mine`, add the following line immediately before the `endm` line:

```
Extens dc.b "os9p2 mine",0
```

- Remake the `Init` module.

Step 3. Create a new bootfile:

- Change to the `BOOTLISTS` directory for your CPU card and edit the bootlist file so the customization module name appears in the bootlist.
- Create a new bootfile with the `os9gen` utility. For example:

```
os9gen /h0fmt -z=bootlist
```

For More Information

`os9gen` builds and links a bootstrap file. Refer to the ***Utilities Reference*** manual for more information about `os9gen`.

Step 4. Reboot the system and make sure the new module is operational.

For More Information

Refer to the ***Using OS-9 for 68K Processors*** manual for additional information on directory structure and making boots. Additional information may also be available in the ***Getting Started*** manual for your system.

Process Creation

All OS-9 programs run as ***processes*** or ***tasks***. The F$Fork system call creates new processes. The name of the primary module the new process is to execute initially is the most important parameter passed in the fork system call. The following outlines the creation process:

1. **Locate or load the program.**
   OS-9 tries to find the module in memory. If it cannot find the module, it loads a mass-storage file into memory using the requested module name as a file name.
2. **Allocate and initialize a process descriptor.**
   After OS-9 locates the primary module, it assigns a data structure called a ***process descriptor*** to the new process. The process descriptor is a table containing information about the process: its state, memory allocation, priority, and I/O paths. The process descriptor is automatically initialized and maintained. The process does not need to know about the descriptor's existence or contents.
3. **Allocate the stack and data areas.**
   The primary module's header contains a data and stack size. OS-9 allocates a ***contiguous memory area*** of the required size from the free memory space. The **Process Memory Areas** section later in this chapter discusses process memory areas.
4. **Initialize the process.**
   OS-9 sets the new process's registers to the proper addresses in the data area and object code module (see **Figure 2-2**). 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 OS-9 cannot perform any of these steps, it aborts the creation of the new process and notifies the process that originated the fork of the error. If OS-9 completes all the steps, it adds the new process to the active process queue for execution scheduling.

The new process is also assigned a ***process ID***. This is a unique number that is the process' identifier. Other processes can communicate with it by referring to its ID in system calls. The process

also has an associated ***group ID*** and ***user ID***. These identify all processes and files belonging to a particular user and group of users. The group and user ID's 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. Terminating the process:

- Closes any open paths.
- Deallocates the process' memory.
- Unlinks its primary module.

Figure 2-2 New Process's Initial Memory Map and Register Contents

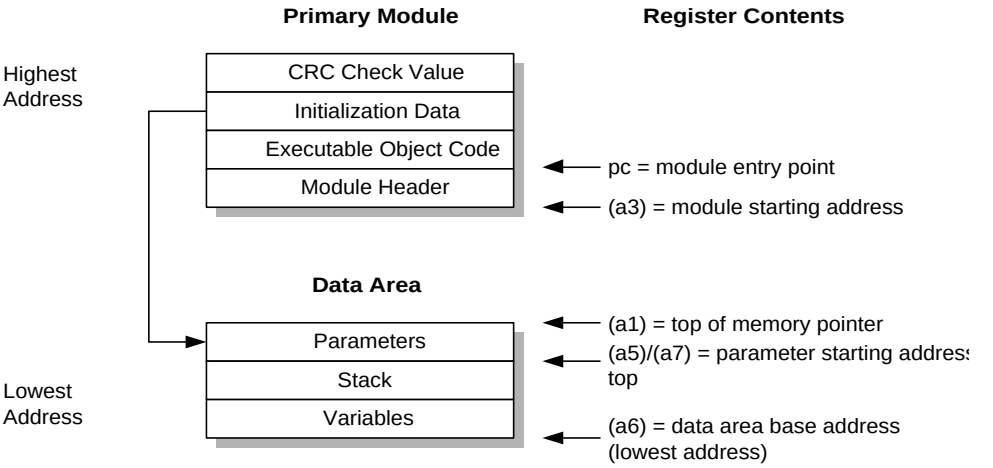


Table 2-6 Registers Passed to the New Process

`sr N000`	`(a0)` undefined
(N=0 for non-MSP systems	`(a1)` top of memory pointer
N=1 for MSP systems)	`(a2)` undefined
`pc` module entry point	`(a3)` primary module pointer

Table 2-6 Registers Passed to the New Process (continued)

d0.w process ID	(a4) undefined
d1.l group/user ID	(a5) parameter pointer
d2.w priority	(a6) static storage (data area) base pointer
d3.w # of paths inherited	(a7) stack pointer (same as a5)
d4.l undefined	
d5.l parameter size	
d6.l total initial memory	

Note

(a6) is always biased by $8000 to allow object programs to access 64K of data using indexed addressing. You can usually ignore this bias because the OS-9 linker automatically adjusts for it.

Process Memory Areas

OS-9 divides all processes (programs) into two logically separate memory areas:

- Code
- Data

This division provides OS-9's modular software capabilities.

Each process has a unique data area, but not necessarily a unique program memory module. This allows two or more processes to share the same copy of a program. This technique is an automatic function of OS-9 and results in extremely efficient use of available memory.

A program must be in the form of an executable memory module to be run. The program is position-independent and ROMable, and the memory it occupies is considered read-only. It may link to and execute code in other modules.

For More Information

Chapter 1: System Overview covers the format of an executable memory module.

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 assigned to the data area are unknown at the time the program is written. A base address is kept in a register (usually a6) to access the data area. You can read or write to this area.

If a program uses variables requiring initialization, OS-9 copies the initial values from the read-only program area to the data area where the variables actually reside. The OS-9 linker builds appropriate initialization tables which initialize the variables.

Process State

A process is either in active, waiting, or sleeping state:

Table 2-7 Process States

State	Description
active	The process is active and ready for execution. The scheduler gives active processes time for execution according to their relative priority with respect to all other active processes. It uses a method that compares the ages of all active processes in the queue. It gives some CPU time to all active processes, even if they have a very low relative priority.
waiting	The process is inactive until a child process terminates or until a signal is received. The process enters the wait state when it executes an `F$Wait` system service request. It remains inactive until one of its descendant processes terminates or until it receives a signal.
sleeping	The process is inactive for a specified period of time or until it receives a signal. A process enters the sleep state when it executes an `F$Sleep` service request or performs an event wait function on a event that is not ready. `F$Sleep` specifies a time interval for which the process is to remain inactive. Processes often sleep 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. Processes waiting on an event are also included in the sleep queue.

There is a separate queue (linked list of process descriptors) for each process state. State changes are made by moving a process descriptor from its current queue to another queue.

Process Scheduling

OS-9 is a multitasking operating system. Two or more independent programs, called ***processes*** or ***tasks***, can execute simultaneously. Several processes share each second of CPU time. Although the processes appear to run continuously, the CPU only executes one instruction at a time. The OS-9 kernel determines which process, and how long, to run based on the priorities of the active processes. ***Task-switching*** is the action of switching from the execution of one process to another. Task-switching does not affect the programs' execution.

A real-time clock interrupts the CPU at every ***tick***. By default, a tick is .01 second (10 milliseconds). At any occurrence of a tick, OS-9 can suspend execution of one program and begin execution of another. The tick length is hardware dependent. Thus, to change the tick length, you must rewrite the clock driver and re-initialize the hardware.

A ***slice*** or ***timeslice*** is the longest amount of time a process controls the CPU before the kernel re-evaluates the active process queue. By default, a slice is two ticks. You can change the number of ticks per slice by adjusting the system global variable `D_TSlice` or by modifying the `Init` module.

To ensure efficiency, only processes on the active process queue are considered for execution. The active process queue is organized by ***process age***, a count of how many task switches have occurred since the process entered the active queue plus the process's initial priority. The oldest process is at the head of the queue. OS-9's scheduling algorithm allocates some execution time to each active process.

When a process is placed in the active queue, its age is set to the process's assigned priority and the ages of all other processes increment. Ages never increment beyond $ffff.

After the currently executing process's timeslice, the kernel executes the process with the highest age.

Preemptive Task-Switching

During critical real-time applications you sometimes need fast interrupt response time. OS-9 provides this by preempting the currently executing process when a process with a higher priority becomes active. The lower priority process loses the remainder of its time-slice and is re-inserted into the active queue.

Two system global variables affect task-switching:

- `D_MinPty` (minimum priority)
- `D_MaxAge` (maximum age)

Both variables are initially set in the `Init` module. Users with a group ID of zero (super users) can access both variables through the `F$SetSys` system call.

For More Information

`F$SetSys` allows you to set and examine OS-9 system global variables. Refer to Chapter 1: System Overview for more information about `F$SetSys`.

D_MinPty: Specifying a Minimum Priority

If a process' priority or age is less than `D_MinPty`, the process is not considered for execution and is not aged. Usually, this variable is not used; it is set to `0`.

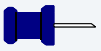
WARNING

If the minimum system priority is set above the priority of all running tasks, the system is completely shut down. You must reset to recover. Therefore, it is crucial to restore `D_MinPty` to a normal level when the critical task(s) finishes.

D_MaxAge: Specifying a Maximum Age

`D_MaxAge` is the maximum age to which a process can increment. When `D_MaxAge` is activated, tasks are divided into two classes:

High priority	tasks receive all of the available CPU time and do not age.
Low priority	tasks do not age past `D_MaxAge`. Low priority tasks are run only when the high priority tasks are inactive. Usually, this variable is not used; it is set to `0`.



Note

System state processes may or may not be time-slicable, depending on the process's requirements and/or the system configuration control, as follows:

- If the `Disable SysState Time Slice` flag is set in the `M$SysConf` field of the `Init` module, the system state time slicing is disabled for the entire system. This option is provided primarily for backwards compatibility with V2.N system state modules, as these may be executing critical routines affecting shared system resources.

- If the M$SysConf flag allows system state time-slicing, and if the process has sections of code that can not be time-sliced, the process must do one of the following prior to entering the first critical region:

  a. Set the P$Preempt field of its process descriptor to a non-zero value, so as to disable the process's ability to be time-sliced, or

  b. Increment/decrement the P$Preempt field on entry/exit to critical (non-preemptable) code sections. This second method is the preferred method, as it allows the process to be time-sliced whenever possible.

Exception and Interrupt Processing

One of OS-9's features is its extensive support of the 68K family advanced exception/interrupt system. You can install routines to handle particular exceptions using various OS-9 system calls for the types of exceptions.

Table 2-8 Vector Descriptions for 68000/008/010/070/CPU32 Family

Vector Number	Related OS-9 Call	Assignment
0	none	Reset initial Supervisor Stack Pointer (SSP)
1	none	Reset initial Program Counter (PC)
2	F$STrap	Bus error
3	F$STrap	Address error
4	F$STrap	Illegal instruction
5	F$STrap	Zero divide
6	F$STrap	CHK instruction; CHK2 (CPU32)
7	F$STrap	TRAPV instruction
8	F$STrap	Privilege violation
9	F$DFork	Trace
10	F$STrap	Line 1010 emulator
11	F$STrap	Line 1111 emulator

Table 2-8 Vector Descriptions for 68000/008/010/070/CPU32 Family (continued)

Vector Number	Related OS-9 Call	Assignment
12	none	Reserved (000/008/010/070); hardware break point (CPU32)
13	none	Reserved
14	none	Reserved (000/008); format error (010/070/CPU32)
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$TLink	User TRAP #1-15 instruction vectors
48-56	none	Reserved
57-63	none/ F$IRQ	Reserved (000/008/010/CPU32) on-chip level 1-7 auto-vectored interrupts (070)
64-255	F$IRQ	Vectored interrupts (user defined)

Table 2-9 Vector Descriptions for 68020/030/040

Vector Number	Related OS-9 Call	Assignment
0	none	Reset initial Supervisor Stack Pointer (SSP)
1	none	Reset initial Program Counter (PC)
2	F$STrap	Bus error
3	F$STrap	Address error
4	F$STrap	Illegal instruction
5	F$STrap	Zero divide
6	F$STrap	CHK, CHK2
7	F$STrap	TRAPV cpTRAPcc, TRAPcc
8	F$STrap	Privilege violation
9	F$DFork	Trace
10	F$STrap	Line 1010 emulator
11	F$STrap	Line 1111 emulator
12	none	Reserved
13	none	Coprocessor protocol violation (020,030 only); reserved (040)
14	none	Format error

Table 2-9 Vector Descriptions for 68020/030/040 (continued)

Vector Number	Related OS-9 Call	Assignment
15	none	Uninitialized interrupt
16-23	none	Reserved
24	none	Spurious interrupt
25-31	F$IRQ	Level 1-7 interrupt autovectors
32	F$OS9	User TRAP #0 instruction (OS-9 call)
33-47	F$TLink	User TRAP #1-15 instruction vectors
48	F$STrap	FPCP Branch, or set on unordered condition
49	F$STrap	FPCP Inexact result
50	F$STrap	FPCP Divide by zero
51	F$STrap	FPCP Underflow error
52	F$STrap	FPCP Operand error
53	F$STrap	FPCP Overflow error
54	F$STrap	FPCP NAN signaled
55	F$STrap	Reserved (020/030); FPCP Unimplemented data type (040)
56	none	PMMU Configuration (020/030); reserved (040)

Table 2-9 Vector Descriptions for 68020/030/040 (continued)

Vector Number	Related OS-9 Call	Assignment
57	none	PMMU Illegal Operation (020); reserved (030/040)
58	none	PMMU Access Level Violation (020); reserved (030/040)
59-63	none	Reserved
64-255	F$IRQ	Vectored interrupts (user defined)

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 at least 4K of RAM below and 4K of RAM above this address for system global storage. Each time an exception occurs, OS-9 uses this vector to find the base address of system global data.

The reset initial program counter (PC) is the coldstart entry point to OS-9. After startup, its only use is to reset after a catastrophic failure.

WARNING

User programs should not use or modify either of these vectors.

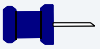
Error Exceptions: vectors 2 - 8, 10 - 24, 48 - 63

These exceptions are usually considered fatal program errors and cause a user program to unconditionally terminate. If F$DFork created the process, the process resources remain intact and control returns to the parent debugger to allow a postmortem examination.

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

When an error exception occurs, the user subroutine executes 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 was passed bad data and an error is returned. In some cases, system data structures are damaged by passing nonsense parameters to system calls.



Note

Not all catchable exception vectors are applicable to all 68000-family CPUs. For example, vectors 48-54 (FPCP exceptions) only apply to 68020 and 68030 CPUs.

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.

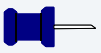
These exceptions provide interrupt polling for I/O devices that do not generate vectored interrupts. Internally, they are handled exactly like vectored interrupts.

WARNING

Normally, you should not use Level 7 interrupts because they are non-maskable and can interrupt the system at dangerous times. You may use Level 7 interrupts for software refresh of dynamic RAMs or similar functions provided the IRQ service routine does not use any OS-9 system calls or system data structures.

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 executes whenever the user trap is called. In addition, trap handlers have initialization and termination entry points that execute when linked and at process termination. The termination entry point is not currently implemented.



Note

Trap 13 (CIO) and trap 15 (math) are standard trap handlers distributed by Microware.

The 192 vectored interrupts provide a minimum amount of system overhead in calling a device driver module to handle an interrupt. Interrupt service routines execute in system state without an 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, although this entry point is not currently implemented. The FIRQ/FFIRQ system calls install a handler in the system's interrupt tables. If necessary, multiple devices may be used on the same vector.

Chapter 3: OS-9 Input/Output System

This chapter explains the software components of the OS-9 I/O system and the relationships between those components. It includes the following topics:

- **The OS-9 Unified Input/Output System**
- **The Kernel and I/O**
- **IOMan and I/O**
- **File Managers**
- **File Manager Organization**

The OS-9 Unified Input/Output System

OS-9 allows the choice of the I/O system to be used on the target system. The `Init` module's `M$IOMan` field specifies the name of the I/O module to be used (if any).

The standard I/O module for OS-9 is called IOMan. IOMan provides the basis of the unified I/O system for OS-9, and is required for any system using standard OS-9 file managers (for example, RBF, SCF, and device drivers).

IOMan is provided in two versions:

- Development kernel environments (`IOMan_DEV`).
  This version performs parameter validation. For example, verify user's buffer is actually allocated to the user for a Read system call.
- Atomic kernel environments (`IOMan_ATOM`).
  This version omits the checks the Development kernel performs.

In addition to using the standard IOMan module for the I/O system, system integrators also have the option of either:

- Designing their own custom I/O system and using that module as the replacement for IOMan, or
- Having no formal I/O system and performing I/O directly from applications themselves.

The replacement or removal of IOMan typically occurs on embedded, Atomic kernel environments, where resource and performance requirements are critical, or where the overhead of a formal I/O system is not required.

Note

If the IOMan module is removed or replaced, you should carefully check Appendix D: OS-9 for 68K System Calls where the system calls supported by IOMan are detailed.

Note

The following discussions regarding OS-9's unified I/O system assume IOMan (or a full functional equivalent) is being used in the system.

The OS-9 I/O system is modular; you can easily expand or customize it. It is a versatile, unified, hardware-independent I/O system and consists of the following software components:

- IOMan
- File managers
- Device drivers
- Device descriptor

IOMan, file managers, and device drivers process I/O service requests at different levels. The device descriptor contains information used to assemble the elements of a particular I/O subsystem. The file manager, device driver, and device descriptor modules are standard memory modules. You can install or remove any of these modules while the system is running.

IOMan Overview

IOMan supervises the overall OS-9 I/O system. IOMan:

- **Maintains the I/O modules by managing various data structures.**
  IOMan ensures the appropriate file manager and device driver modules process each I/O request.
- **Establishes paths.**
  The paths connect the kernel, IOMan, the application, the file manager, and the device driver.

File Manager Overview

File managers perform the processing for a particular class of devices, such as disks or terminals. They deal with ***logical*** operations on the class of devices. For example, the Random Block File manager (RBF) maintains directory structures on disks; the Sequential Character File manager (SCF) edits the data stream it receives from terminals.

Note

File managers deal with the I/O requests on a generic ***class*** basis.

The Kernel and I/O

The kernel provides the connection between the I/O system and the application. When an application makes an I/O (`I$`) service request, it is identified as such by the kernel performing the following:

- Disables system state time-slicing for the process (the file manager can later re-enable this if desired)
- Calls the appropriate entry point in IOMan
- On return from IOMan, re-enables system state time-slicing and returns any error result to the application

Device Driver Overview

Device drivers operate on a class of hardware. Operating on the actual hardware device, they send data to and from the device on behalf of the file manager. They isolate the file manager from hardware dependencies such as control register organization and data transfer modes, translating the file manager's logical requests into specific hardware operations.

Device Descriptor Overview

The device descriptor contains the information required to assemble the various I/O subsystems (device components). A device descriptor contains the names of the file manager and device driver associated with the device, as well as the device's operating parameters. Parameters in device descriptors can be:

- Fixed, such as interrupt level and port address.
- Variable, such as terminal editing settings and disk physical parameters.

The variable parameters in device descriptors provide the initial default values when a path is opened, but applications can change these values. The device descriptor name is the name of a device as known by the user. For example, the device `/d0` is described by the device descriptor `d0`.

IOMan and I/O

IOMan manages I/O system calls by routing data between processes and the appropriate file managers and device drivers. IOMan also allocates and initializes global static storage for device drivers.

IOMan maintains two important internal data structures:

- The ***device table***
- The ***path table***

These tables reflect two other structures respectively:

- The ***device descriptor***
- The ***path descriptor***

When a path is opened, IOMan attempts to link to the device descriptor associated with the device name specified (or implied) in the pathlist. The device descriptor contains the names of the device driver and file manager for the device. The information in the device descriptor is saved by IOMan in the device table so it can route subsequent system calls to these modules.

Paths maintain the status of I/O operations to devices and files. IOMan maintains these paths using the path table. Each time a path is opened, a path descriptor is created and an entry is added to the path table. When a path is closed, the path descriptor is deallocated and its entry is deleted from the path table.

When an `I$Attach` system call is first performed on a new device descriptor, the kernel creates a new entry in the device table. Each entry in the table has specific information from the device descriptor concerning the appropriate file manager and driver. It also contains a pointer to the device driver static storage. For each device in the table, IOMan maintains a use count indicating the current number of device users.

For More Information

`I$Attach` attaches a new device to the system. Refer to **Appendix D: OS-9 for 68K System Calls** for more information about `I$Attach`.

Device Descriptor Modules

Device descriptor modules are small, non-executable modules containing information to associate a specific I/O device with its:

- Logical name
- Hardware controller address(es)
- Device driver name
- File manager name
- Initialization parameters

File managers operate on a class of ***logical*** devices. Device drivers, on the other hand, operate on a class of ***physical*** devices. A device descriptor module tailors a device driver or file manager to a specific I/O port. At least one device descriptor module must exist for each I/O device in the system. An I/O device may have several device descriptors with different initialization parameters and names. For example, a serial/parallel driver could have two device descriptors, one for terminal operation (`/t1`) and one for printer operation (`/p1`).

If a suitable device driver exists, you can add devices to the system by adding the new hardware and another device descriptor. While the system is running, device descriptors can be:

- In ROM
- In the boot file
- Loaded into RAM

The module name is used as the logical device name by the system and user (it is the device name given in pathlists). Its format consists of a standard module header with a type code of device descriptor (`DEVIC`). The remaining module header fields are shown in **Table 3-2**.

Note

These fields are standard for all device descriptor modules. They are followed by a device-specific initialization table. Appendix B: Path Descriptors and Device Descriptors contains the initialization tables for each standard class of I/O devices (RBF, SCF, and SBF).

Table 3-1 Device Descriptor Modules

Name	Description
`M$Port`	**Port Address** The absolute physical address of the hardware controller.
`M$Vector`	**Interrupt Vector Number** The interrupt vector associated with the port, used to initialize hardware and for installation on the IRQ poll table: 25-31 for an auto-vectored interrupt. Levels 1-7. 57-63 for 68070 on-chip auto-vectored interrupts. Levels 1-7. 64-255 for a vectored interrupt.

Table 3-1 Device Descriptor Modules (continued)

Name	Description
`M$IRQLvl`	**Interrupt Level** The device's physical interrupt level. It is ***not*** used by the kernel or file manager. The device driver may use it to mask off interrupts for the device when critical hardware manipulation occurs.
`M$Prior`	**Interrupt Polling Priority** Indicates the priority of the device on its vector. Smaller numbers are polled first if more than one device is on the same vector. A priority of `0` indicates the device requires exclusive use of the vector.
`M$Mode`	**Device Mode Capabilities** This byte is used to validate a caller's access mode byte in `I$Create` or `I$Open` calls. If the bit is set, the device can perform the corresponding function. If the `Share_bit` (single user bit) is set here, the device is non-sharable. This is useful for printers.
`M$FMgr`	**File Manager Name Offset** The offset to the name string of the file manager module for this device.
`M$PDev`	**Device Driver Name Offset** The offset to the name string of the device driver module for this device.

Table 3-1 Device Descriptor Modules (continued)

Name	Description
M$DevCon	**Device Configuration** The offset to an optional device configuration table. You can use it to specify parameters or flags the device driver needs that are not part of the normal initialization table values. This table is located after the standard initialization table. The kernel or file manager never references it. As the pointer to the device descriptor is passed in INIT and TERM, M$DevCon is generally available to the driver only during the driver's INIT and TERM routines. Other routines in the driver (for example, Read) must first search the device table to locate the device descriptor before they can access this field. Typically, this table is used for name string pointers, OEM global allocation pointers, or device-specific constants/flags. **NOTE:** These values, unlike the standard options, are not copied into the path descriptor's options section.
M$DevFlags	**Device Flags** Reserved for future use.

Table 3-1 Device Descriptor Modules (continued)

Name	Description
M$Opt	**Table Size** This contains the size of the device's standard initialization table. Each file manager defines a ceiling on M$Opt.
M$DTyp	**Device Type (first field of initialization table)** The file manager associated with the device defines the device's standard initialization table, with the exception of the first byte (M$DTyp). The first byte indicates the class of the device (for example, RBF or SCF). The initialization table (M$DTyp through M$DTyp + M$Opt) is copied into the option section of the path descriptor when a path to the device is opened. Typically, this table is used for the default initialization parameters such as the delete and backspace characters for a terminal. Applications may examine all of the values in this table using I$GetStt (SS_Opt). Some of the values may be changed using I$SetStt; some are protected by the file manager to prevent inappropriate changes. The theoretical maximum initialization table size is 128 bytes. However, a file manager may restrict this to a smaller value.

Note

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`.

Table 3-2 Additional Standard Header Fields for Device Descriptors

Offset	Name	Description
$30	M$Port	Port Address
$34	M$Vector	Trap Vector Number
$35	M$IRQLvl	IRQ Interrupt Level
$36	M$Prior	IRP 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
$40	M$DevFlags	Device Flags (reserved)
$44		Reserved

Table 3-2 Additional Standard Header Fields for Device Descriptors

Offset	Name	Description
$46	M$Opt	Initialization Table Size
$48	M$DTyp	Device Type

Adding Additional Devices

You can add additional devices to your system. If an identical device controller already exists, you only need to 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***. A path descriptor contains information required by file managers and device drivers to perform I/O functions. Path descriptors are dynamically allocated and de-allocated as paths are opened and closed.

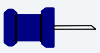
Path descriptors have three sections:

- The first 30 bytes are defined universally for all file managers and device drivers.
- PD_FST is reserved for and defined by each type of file manager for items such as file pointers, and permanent variables.

- PD_OPT is a 128-byte option area used for dynamically alterable operating parameters for the file or device. These variables are initialized when the path is opened by copying the initialization table contained in the device descriptor module, and can be examined or altered later by user programs via GetStat and SetStat system calls. Not all options can be modified.

For More Information

Refer to Appendix B: Path Descriptors and Device Descriptors for the current definitions of the path descriptor option area for each standard class of I/O devices (RBF, SCF, SBF, and Pipes). The definitions are included in sys.l or usr.l, and are linked into programs requiring them.



Note

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.

Table 3-3 Universal Path Descriptor Definitions

Offset	Name	Maintained By	Description
$00	PD_PD	IOMan	Path Number
$02	PD_MOD	IOMan	Access Mode (R W E S D)

Table 3-3 Universal Path Descriptor Definitions (continued)

Offset	Name	Maintained By	Description
$03	PD_CNT	IOMan	Number of Paths using this PD (obsolete)
$04	PD_DEV	IOMan	Address of Related Device Table Entry
$08	PD_CPR	IOMan	Requester's Process ID
$0A	PD_RGS	IOMan	Address of Caller's MPU Register Stack
$0E	PD_BUF	File Manager	Address of Data Buffer
$12	PD_USER	IOMan	Group/User ID of Original Path Owner
$16	PD_PATHS	IOMan	List of Open Paths on Device
$1A	PD_COUNT	IOMan	Number of Paths using this PD
$1C	PD_LProc	IOMan	Last Active Process ID
$20	PD_ErrNo	File Manager	Global ***errno*** for C language file managers
$24	PD_SysGlob	File Manager	System global pointer for C language file managers

Table 3-3 Universal Path Descriptor Definitions (continued)

Offset	Name	Maintained By	Description
$2A	PD_FST	File Manager	File Manager Working Storage
$80	PD_OPT	Driver/ File Manager	Option Table

File Managers

File managers process the raw data stream to or from device drivers for a class of similar devices. File managers make device drivers 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. They 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.

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.

Microware includes the following file managers in the OS-9 standard distributions.

Note

I/O system modules must:

- Be owned by a super-user (`0.n`).
- Have the system-state bit set in the attribute byte of the module header. OS-9 does not currently make use of this, but future revisions may require I/O system modules be system-state modules.

Embedded

Table 3-4 Embedded File Managers

Name	Description
SCF	**Sequential Character File Manager** Handles I/O for sequentially character- structured devices, such as terminals, printers, and modems.
PIPEMAN	**Pipe File Manager** Supports interprocess communications through memory buffers called ***pipes***.

Disk Based

Table 3-5 Disk Based File Managers

Name	Description
RBF	**Random Block File Manager** Handles I/O for random-access, block-structured devices, such as floppy/hard disk systems.
SBF	**Sequential Block File Manager** Handles I/O for sequentially block- structured devices, such as tape systems.
PCF	**PC File Manager** Handles reading/writing PC-DOS disks. It uses RBF drivers.

Extended

Table 3-6 Extended File Managers

Name	Description
IFMAN	**Communications Interface File Manager** Manages network interfaces.
PKMAN	**Pseudo-Keyboard File Manager** Provides an interface to the driver side of SCF to enable the software to emulate a terminal.
SOCKMAN	**Socket File Manager** Creates and manages the interface to communication protocols (sockets).
NFS	**Network File System Manager** Client file manager for mounting remote file systems. The NFS protocol provides remote access to shared file systems over local area networks.

Other

Microware also supports the following file managers that are not included in the OS-9 distribution:

Table 3-7 Other Microware File Managers

Name	Description
CDFM	**Compact Disc File Manager** Handles CD and audio devices, as well as access to CD ROM and CD audio.
UCM	**User Communications Manager** Handles video, pointer, and keyboard devices for CD-I (Compact Disc-Interactive).
GFM	**Graphics File Manager** Provides a full set of text and graphics primitives, input handling for keyboards and pointers, and high level features for handling user interaction in a real-time, multitasking environment.
NFM	**Network File Manager** Processes data requests over the OS-9 network.
NRF	**Non-Volatile RAM File Manager** Controls non-volatile RAM and handles a flat (non-hierarchical) directory structure.
ISM	**ISDN Basic Rate Interface Manager.** Manager connections for Basic Rate (2B+D) Interfaces to the Integrated Services Digital Network (ISDN).

File Manager Organization

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 execution entry point offset in the module header specifies the location of the table. These routines are called in system state. A sample listing of the beginning of a file manager module is listed in **Beginning of a Sample File Manager Module**.

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

Table 3-8 Registers

Register	Pointer to the
(a1)	Path descriptor.
(a4)	Current process descriptor.
(a5)	User's register stack. User registers pass/receive parameters as shown in the system call description section.
(a6)	System global data area.

Beginning of a Sample File Manager Module

```
* Sample File Manager
* Module Header declaration
    Type_Lang equ (FlMgr<<8)+Objct
    Attr_Revs equ ((ReEnt+Supstat)<<8)+0
    psect FileMgr,Type_Lang,Attr_Revs,Edition,0,Entry_pt
* Entry Offset Table
Entry_pt dc.w Create-Entry_pt
         dc.w Open-Entry_pt
         dc.w MakDir-Entry_pt
         dc.w ChgDir-Entry_pt
```

```
        dc.w Delete-Entry_pt
        dc.w Seek-Entry_pt
        dc.w Read-Entry_pt
        dc.w Write-Entry_pt
        dc.w ReadLn-Entry_pt
        dc.w WriteLn-Entry_pt
        dc.w GetStat-Entry_pt
        dc.w SetStat-Entry_pt
        dc.w Close-Entry_pt
* Individual Routines Start Here
```

File Manager I/O Responsibilities

The following are the file manager's I/O responsibilities:

Table 3-9 File Manager I/O Responsibilities

Name	Description
`Open`	Open 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 multifile devices (such as RBF and PIPEMAN), directory searching is performed to find the specified file.
`Create`	Perform the same function as `Open`. If the file manager controls multifile devices (such as RBF and PIPEMAN), a new file is created.
`Makdir`	Create a directory file on multifile devices. `Makdir` is neither preceded by a `Create`, nor followed by a `Close`. File managers that cannot support directories return with the carry bit set and an appropriate error code in register `d1.w`.

Table 3-9 File Manager I/O Responsibilities (continued)

Name	Description
`Chgdir`	On multifile devices, `ChgDir` searches for a directory file. If the directory is found, the address of the directory is saved in the caller's process descriptor at `P$DIO`. The kernel allocates a path descriptor so the `ChgDir` function may save information about the directory file for later searching. `Open` and `Create` begin searching in this directory when the caller's pathlist does not begin with a slash (`/`) character. File managers that do not support directories return with the carry bit set and an appropriate error code in register `d1.w`.
`Delete`	Multifile device managers usually do a directory search similar to `Open` and, once found, remove the file name from the directory. Any media the file was using is returned to unused status. File managers that do not support multifile devices return an `E_UNKSVC` error.
`Seek`	File managers that support random access devices use `Seek` to position file pointers of the already open path to the specified byte. 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 `E_UNKSVC` error.

Table 3-9 File Manager I/O Responsibilities (continued)

Name	Description
`Read`	Return the number of bytes requested to the user's data buffer. If there is no data available, an `EOF` error is returned. `Read` must be able to copy pure binary data. It generally does not edit the data. Usually, the file manager calls 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`	`Write`, like `Read`, must be able to record pure binary data without alteration. Usually, the `Read` and `Write` routines are nearly identical. The most notable difference is `Write` uses the device driver's output routine instead of the input routine. Writing past the end of file on a device expands the file with new data. RBF and similar random access devices that use fixed-length records (sectors) must often pre-read a sector before writing it unless the entire sector is being written.
`Readln`	`ReadLn` differs from `Read` in two respects. First, `ReadLn` is expected to terminate when the first end-of-line character (carriage return) is encountered. Second, `ReadLn` performs any input editing appropriate for the device. Specifically, the SCF file manager performs editing that involves functions such as handling backspace, line deletion, and echo.

Table 3-9 File Manager I/O Responsibilities (continued)

Name	Description
`Writeln`	`Writeln` is the counterpart of `Readln`. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing also is performed. After a carriage return, for example, SCF usually outputs a line feed character and nulls (if appropriate).
`GetStat`	`GetStat` (Get Status) is a wildcard call designed to provide the status of various features of a device (or file manager) that are not generally device independent. The file manager may perform some specific function such as obtaining the size of a file. Status calls that are unknown to the file manager are passed to the driver to provide a further means of device independence.
`SetStat`	`SetStat` (Set Status) is similar to `GetStat`. However, `SetStat` is generally used to set the status of various features of a device (or file manager). 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 to the file manager are passed to the driver to provide a further means of device independence. For example, a `SetStat` call to format a disk track may behave differently on different types of disk controllers.
`Close`	`Close` ensures any output to a device is completed (writing out the last buffer if necessary), and releases any buffer space allocated when the path was opened. It may do specific end-of-file processing if necessary, such as writing end-of-file records on tapes.

Device 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 or directories, which are handled at a higher level by the OS-9 file manager. Because device driver modules are re-entrant, one copy of the module can simultaneously support multiple devices using 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. To present this information in an understandable manner, only basic drivers for character-oriented (SCF-type) and disk-oriented (RBF-type) devices are covered.

If written properly, a single physical driver module can handle multiple identical hardware interfaces. The specific information for each physical interface (such as port address and initialization constants) is provided in the 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 initialization data of the device descriptor to the path descriptor data structure for easy access by the drivers.

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

- Initialize the device controller hardware and related driver variables as required.
- Read a standard physical unit (a character or sector, depending on the device type).
- Write a standard physical unit (a character or sector, depending on the device type).
- Return a specified device status.
- Set a specified device status.
- Deinitialize the device. It is assumed the device will not be used again unless reinitialized.
- Process an error exception generated during driver execution.

The interrupt service subroutine is also part of the device driver, although it is not called by the file manager, but by the kernel's interrupt routine. It communicates with the driver's main section through the static storage and certain system calls.

Driver Module Format

All drivers must conform to the standard OS-9 memory module format. The module type code is `Drivr`. Drivers should have the system-state bit set in the attribute byte of the module header. Currently OS-9 does not make use of this, but future revisions may require all device drivers to be system state modules. A sample assembly language header is shown in **Sample Driver Module Header Format**.

The execution offset in the module header (`M$Exec`) gives the address of an ***offset table.*** The offset table specifies the starting address of each of the seven driver subroutines relative to the base address of the module.

The static storage size (`M$Mem`) specifies the amount of local storage the driver requires. This is the sum of the global I/O storage, the storage required by the file manager (`V_xxx` variables), and any variables and tables declared in the driver.

The driver subroutines are called by the associated file manager through the offset table. The driver routines are always executed in system state. Regardless of the device type, the standard parameters listed here are passed to the driver in registers. You may also pass other parameters that depend on the device type and subroutine called. These are described in individual chapters concerning the file managers in the ***OS-9 for 68K Processors Technical I/O Manual***.

INITIALIZE and TERMINATE

Table 3-10 Registers Used to Initialize and Terminate

Register	Address of the
(a1)	Device descriptor module.
(a2)	Driver's static variable storage.
(a4)	Process descriptor requesting the I/O function.
(a6)	System global variable storage area.

READ, WRITE, GETSTAT, and SETSTAT

Table 3-11 Registers Used to Read, Write, Getstat, and Setstat

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

ERROR

You should define this entry point as the offset to the error exception handling code or zero if no handler is available. The kernel does not currently use this entry point. However, this entry point may be accessed in future revisions.

An `rts` instruction terminates each subroutine. Error status is returned using the CCR carry bit with an error code returned in register `d1.w`.

Sample Driver Module Header Format

```
* Module Header
Type_Lang equ (Drivr<<8)+Objct
Attr_Revs equ ((ReEnt+Supstat)<<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 (0=none)
```

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

For character-oriented devices, set up the controller to generate an interrupt when an incoming character is received and when the transmission of an out-going character is completed. The driver should buffer both the input data and the output data.

In the case of block-type devices (for example, RBF or SBF), set up the controller to generate an interrupt when a block read or write operation finishes. The driver does not need to buffer data because the driver is passed the address of a complete buffer. Direct Memory Access (DMA) transfers, if available, significantly improve data transfer speed.

Usually, the Init routine adds the relevant device interrupt service routine to the OS-9 interrupt polling system using the F$IRQ and/or F$FIRQ system call. The READ and WRITE routines enable and disable the controller interrupts as required. TERM disables the physical interrupts and then takes the device off the interrupt polling table.

The assignment of device intercept priority levels can have a significant impact on system operation. Generally, the smarter the device, the lower you can set its interrupt level. For example, a disk controller that buffers sectors can wait longer for service than a single-character buffered serial port. Assign the clock tick device the highest possible level to keep system time-keeping interference at a minimum.

The following table shows how you can assign interrupt levels:

```
level  6: clock ticker
   5: "dumb" (non-buffering) disk controller
   4: terminal port
   3: printer port
   2: "smart" (sector-buffering) disk controller
```

Chapter 4: Interprocess Communications

This chapter describes the five forms of interprocess communication supported by OS-9. It includes the following topics:

Introduction

This chapter describes the six forms of interprocess communication OS-9 supports:

Table 4-1 Forms of Interprocess Communication

Name	Description
Signals	Synchronize concurrent processes.
Alarms	Send signals or execute subroutines at specified times.
Events	Synchronize the access of shared resources for concurrent processes.
Semaphores	Support exclusive access to shared resources.
Pipes	Transfer data among concurrent processes.
Data Modules	Transfer or share data among concurrent processes.

Signals

In interprocess communications, a ***signal*** is an intentional disturbance in a system. OS-9 signals are designed to synchronize concurrent processes, but you can also use them to transfer small amounts of data.

Note

Because they are usually processed immediately, signals provide real-time communication between processes.

Signals are also referred to as ***software interrupts*** because a process receives a signal similar to a CPU receiving an interrupt. Signals enable a process to send a ***numbered interrupt*** to another process.

If an active process receives a signal:

- The intercept routine executes immediately (if installed)
- The process resumes execution where it left off

If a sleeping or waiting process receives a signal:

- The process moves to the active queue
- The intercept handler executes
- The process resumes execution immediately after the call that removed it from the active queue

Note

If a process receives a signal for which it does not have an intercept routine, the process is killed. This applies to all signals greater than 1, which is the wake-up signal.

When you send a signal, it has two parameters:

- The process ID of the destination
- A signal code

Supported User-State Signal Codes

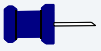
OS-9 supports the following signal codes in user-state:

Table 4-2 User-state Signal Codes

Signal	Description
`0`	**Unconditional Process Abort Signal** The super-user can send the kill signal to any process, but non-super-users can send this signal only to processes with their group and user IDs. This signal terminates the receiving process regardless of the state of its signal mask, and is not intercepted by the intercept handler.
`1`	**Wake-Up Signal** Sleeping/waiting processes receiving this signal are awakened, but the intercept handler does not intercept the signal. Active processes ignore this signal. The wake-up signal is not queued if the process' signals are masked.
`2-31`	**Deadly I/O Signals:**
`2`	**Keyboard Abort Signal** Entering `<control>E` sends this signal to the last process to perform I/O on the terminal. Usually, the intercept routine performs `exit(2)` when it receives a keyboard abort signal.

Table 4-2 User-state Signal Codes (continued)

Signal	Description
3	**Keyboard Interrupt Signal** Entering <control>C sends this signal to the last process to perform I/O on the terminal. Usually, the intercept routine performs `exit(3)` when it receives a keyboard interrupt signal.
4	**Hang-Up Signal** SCF sends this when the modem connection is lost.
5-19	**Reserved**
20-25	**Reserved**
26-31	**User-Definable Signals** User-definable signals; deadly to I/O operations.
32-65535	**Non-Deadly I/O Signals:**
32-127	**Reserved**
128-191	**Reserved**
192-255	**Reserved**
256-65535	**User-Defined Signals** These signal numbers are non-deadly to I/0 signals.



Note

A program can receive a wake-up signal safely without an intercept handler.

You could design a signal routine to interpret the signal code word as data. For example, you could send various signal codes to indicate different stages in a process' execution. This is extremely effective because signals are processed immediately when received.

The following system calls allow processes to communicate through signals:

Table 4-3 Signal Supported System Calls

Name	Description
F$Send	Send a signal to a process.
F$Icpt	Install a signal intercept routine.
F$Sleep	Deactivate the calling process until the specified number of ticks has passed or a signal is received.
F$Sigmask	Enable/disable signals from reaching the calling process.

For More Information

Appendix A: Example Code contains a program demonstrating how you may use signals.

For specific information about these system calls, refer to **Appendix D: OS-9 for 68K System Calls**. The Microware C compiler supports a corresponding C call for each of these calls as well.

Alarms

User-State Alarms

The user-state F$Alarm (User-State) request allows a program to arrange to send a signal to itself. The signal may be sent:

- At a specific time of day
- After a specified interval passes
- Periodically, each time the specified interval passes

OS-9 supports the following user-state alarm functions:

Table 4-4 User-state Alarm Functions

Name	Description
A$Delete (User-State)	Remove a pending alarm request.
A$Set (User-State)	Send a signal after specified time interval.
A$Cycle (User-State)	Send a signal at specified time intervals.
A$AtDate (User-State)	Send a signal at Gregorian date/time.
A$AtJul (User-State)	Send a signal at Julian date/time.

For More Information

Refer to the description of `F$Alarm (User-State)` in **Appendix D: OS-9 for 68K System Calls** for more information about user-state alarm functions.

Cyclic Alarms

A cyclic alarm is most useful for providing a time base within a program. This makes it easier for you to synchronize certain time-dependent tasks. For example, a real-time game or simulation might allow 15 seconds for each move. You could use a cyclic alarm signal to determine when to update the game board.

The advantages of using cyclic alarms are more apparent when multiple time bases are required. For example, suppose you were using an OS-9 process to update the real-time display of a car's digital dashboard. The process might need to:

- Update a digital clock display every second
- Update the car's speed display five times per second
- Update the oil temperature and pressure display twice per second
- Update the inside/outside temperature every two seconds
- Calculate miles to empty every five seconds

You could give each function the process must monitor a cyclic alarm whose period is the desired refresh rate, and whose signal code identifies the particular display function. The signal handling routine might read an appropriate sensor and directly update the dashboard display. The system takes care of all of the timing details.

Time of Day Alarms

You can set an alarm to provide a signal at a specific time and date. This provides a convenient mechanism for implementing a ***cron*** type of utility which executes programs at specific days and times. Another use is to generate a traditional alarm clock buzzer for personal reminders.

A key feature of this type of alarm is it is sensitive to changes made to the system time. For example, assume the current time is 4:00 and you want a program to send itself a signal at 5:00. The program could either set an alarm to occur at 5:00 or set the alarm to go off in one hour. Assume the system clock is 30 minutes slow, and the system administrator corrects it. In the first case, the program wakes up at 5:00; in the second case, the program wakes up at 5:30.

Relative Time Alarms

You can use a relative time alarm to set a time limit for a specific action. Relative time alarms are frequently used to cause an `I$Read` request to abort if it is not satisfied within a maximum time. To do this:

Step 1. Send a keyboard abort signal at the maximum allowable time.

Step 2. Issue the `I$Read` request.

If the alarm arrives before the input is received, the `I$Read` request returns with an error. Otherwise, the alarm should be cancelled.

For More Information

The **Alarms: Example Program** in Appendix A: Example Code demonstrates this technique.

System-State Alarms

A system-state counterpart exists for each of the user-state alarm functions. However, the system-state version is considerably more powerful than its user-state equivalent:

- When a user-state alarm expires, the kernel sends a signal to the requesting process
- When a system-state alarm expires, the kernel executes the system-state subroutine specified by the requesting process at a very high priority

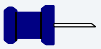
OS-9 supports the following system-state alarm functions:

Table 4-5 System-state Alarm Functions

Name	Description
A$Delete (System-State)	Remove a pending alarm request.
A$Set (System-State)	Execute a subroutine after a specified time interval.
A$Cycle (System-State)	Execute a subroutine at specified time intervals.
A$AtDate (System-State)	Execute a subroutine at a Gregorian date/time.
A$AtJul (System-State)	Execute a subroutine at Julian date/time.

For More Information

Refer to the description of `F$Alarm (System-State)` in Appendix D: OS-9 for 68K System Calls for more information about system-state alarm functions.



Note

The alarm is executed by the kernel's process, not by the original requester's process. During execution, the user number of the system process temporarily changes to the original requester. The stack pointer (a7) passed to the alarm subroutine is within the system process descriptor and contains about 1K of free space.

The kernel automatically deletes a process' pending alarm requests when the process terminates. This may be undesirable in some cases. For example, assume an alarm is scheduled to shut off a disk drive motor if the disk has not been accessed for 30 seconds. The alarm request is made in the disk device driver for the I/O process. This alarm does not work if it is removed when the process exits.

One way to arrange for a persistent alarm is to execute the `F$Alarm (User-State)` request for the system process, rather than the current I/O process. To do this:

Step 1. Move the system variable `D_SysPrc` to `D_Proc`.

Step 2. Execute the alarm request.

Step 3. Restore `D_Proc`.

For example:

```
move.l D_Proc(a6),-(a7)        /*Save current process pointer*/
move.l D_SysPrc(a6),D_Proc(a6) /*Impersonate system process*/
OS9 F$Alarm                    /*Execute the alarm request*/
                               /*  (error handling omitted)  */
move.l (a7)+,D_Proc(a6)        /*Restore current process*/
```

Note

a6 must be a pointer to the system globals.

WARNING

If you use this technique, you must ensure the module containing the alarm subroutine remains in memory until after the alarm has expired.

An alarm subroutine must not perform any function resulting in any kind of sleeping or queuing. This includes F$Sleep, F$Wait, F$Load, F$Event (wait), F$IOQu, and F$Fork (if it might require F$Load). Other than these functions, the alarm subroutine may perform any task.

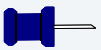
One possible use of the system-state alarm function might be to poll a positioning device, such as a mouse or light pen, every few system ticks. Be conservative when scheduling alarms, and make the cycle as large as reasonably possible. Otherwise, you could waste a great deal of the system's available CPU time.

For More Information

Refer to**Alarms: Example Program** in Appendix A: Example Code for a program demonstrating how you can use alarms.

Events

OS-9 ***events*** are multiple-value semaphores. They synchronize concurrent processes accessing shared resources such as files, data modules, and CPU time. For example, if two processes need to communicate with each other through a common data module, you may need to synchronize the processes so only one updates the data module at a time.



Note

Events do not transmit any information, although processes using the event system may get information about the event and use it as something other than a signaling mechanism.

An OS-9 event is a 32-byte system global variable maintained by the system. Each event includes the following fields:

Table 4-6 Event Fields

Name	Bytes	Description
`Event ID`	2	This number and the event's array position create a unique ID.
`Event name`	12	This name must be unique and cannot exceed eleven characters plus null termination.
`Event value`	4	A value has a range of two billion.

Table 4-6 Event Fields (continued)

Name	Bytes	Description
`Wait increment`	2	This value is added to the event value when a process successfully waits for the event. It is set when the event is created and does not change.
`Signal increment`	2	This value is added to the event value when the event is signaled. It is set when the event is created and does not change.
`Link Count`	2	This is the event use count.
`Next event`	4	This is a pointer to the next process in the event queue. An event queue is circular and includes all processes waiting for the event. Each time the event is signaled, this queue is searched.
`Previous event`	4	This is a pointer to the previous process in the event queue.

The OS-9 event system provides facilities:

- To create and delete events
- To permit processes to link/unlink events and obtain event information
- To suspend operation until an event occurs
- For various means of signaling

You may use events directly as service requests in assembly language programs.

The Wait and Signal Operations

`Wait` and `Signal` are the two most common operations performed on events.

`Wait`	suspends the process until the event is within a specified range, adds the wait increment to the current event value, and returns control to the process just after the wait operation was called.
`Signal`	adds the signal increment to the current event value, checks for other processes to awaken, and returns control to the process.

These operations allow a process to:

- Suspend itself while waiting for an event
- Reactivate when another process signals the event has occurred

For example, you could use events to synchronize the use of a printer. Initialize the event value to `1`, the number of printers on the system. Set the signal increment to `1` and the wait increment to minus one (-1). When a process wants to use the printer, it checks to see if one is available—it waits for the event value to be in the range (`1, number of printers`). In this example, the number of printers is one.

An event value within the specified range indicates the printer is available; the printer is immediately marked as busy (the event value increases by -1, the wait increment) and the process is allowed to use it. An out of range event value indicates the printer is busy and the process is put to sleep on the event queue.

When a process finishes with the printer, the process signals the event; it applies the signal increment to the event value. Then, the event queue is searched for a process whose event value range includes the current event value. If such a process is found, the process:

- Becomes activate
- Applies the wait increment to the event value
- Uses the printer

Coordinating Non-Sharable Resources

To coordinate sharing a non-sharable resource, user programs must:

Step 1. Wait for the resource to become available.

Step 2. Mark the resource as busy.

Step 3. Use the resource.

Step 4. Signal the resource is no longer busy.

The first two steps in this process must be indivisible because of time-slicing. Otherwise, two processes could check an event and find it free. Then, both processes would try to mark it busy. This corresponds to two processes using a printer at the same time. The `F$Event` service request prevents this from happening by performing both steps in the `Wait` operation.

For More Information

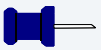
Events: Example Program in Appendix A: Example Code includes a program demonstrating how you may use events.

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 maintenance functions as well as the `Signal` and `Wait` operations. To keep the number of system calls required to a minimum, you can access all event operations through the `F$Event` system call.

Currently, OS-9 has functions to allow you to create, delete, link, unlink, and examine events (listed below). It also provides several variations of the `Signal` and `Wait` operations.



Note

The `F$Event` description in Appendix D: OS-9 for 68K System Calls covers the specific parameters and functions of each event operation. The system definition file `funcs.a` defines `Ev$` function names. Resolve actual values for the function codes by linking with the relocatable library `sys.l` or `usr.l`.

OS-9 supports the following event functions:

Table 4-7 Event Functions

Name	Description
`Ev$Link`	Link to an existing event by name.
`Ev$UnLnk`	Unlink an event.
`Ev$Creat`	Create a new event.
`Ev$Delet`	Delete an existing event.
`Ev$Wait`	Wait for an event to occur.
`Ev$WaitR`	Wait for a relative event to occur.
`Ev$Read`	Read an event value without waiting.

Table 4-7 Event Functions (continued)

Name	Description
Ev$Info	Return event information.
Ev$Pulse	Signal an event occurrence and search waiting processes. Temporarily changes the event value.
Ev$Signl	Signal an event occurrence and search waiting processes. Changes the event value.
Ev$Set	Set an event variable, signal an event occurrence, and search waiting processes.
Ev$SetR	Set an event variable relative to its current value and search the waiting processes.

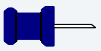
Semaphores

Semaphores support exclusive access to shared resources. Semaphores are similar to events in the way in which they provide applications with mutually exclusive access to data structures. Semaphores differ from events in they are strictly binary in nature which increases their efficiency.

OS-9 supports the following semaphore routines:

Table 4-8 Semaphore Routines

Name	Description
`_os_sema_init()`	Initialize the semaphore data structure for use.
`_os_sema_p()`	Reserve a semaphore.
`_os_sema_term()`	Terminate the use of a semaphore data structure.
`_os_sema_v()`	Release a semaphore.



Note

Using C bindings is the preferred method of accessing OS-9 semaphores.

A single semaphore system call, F$Sema, provides all of the semaphore's functionality. F$Sema requires two parameters:

- One indicating which operation is being performed on the semaphore
- A pointer to the semaphore structure

Unlike events, there is no system call provided to create a semaphore. You must provide the storage for the semaphore. Because semaphores are typically used to protect specific resources, you should declare the semaphore structure as part of the resource structure.

A typical application using semaphores would create a data module containing the memory for the intended resource and its associated semaphore. By using a data module for implementing semaphores, applications can use the OS-9's module protection mechanisms to protect the semaphore.

For More Information

Semaphores: Example Program in Appendix A: Example Code includes a program demonstrating how you may use semaphores.

Once you have created and initialized the semaphore data module, additional processes within the application may use the semaphore by linking to the semaphore data module. You must create the semaphore data module with appropriate permissions to allow the other processes within the application to link to and use the semaphore and its resource.

Semaphore States

A semaphore has two states:

Reserved	any process attempting to reserve the semaphore waits, including the process reserving the semaphore.
Free	any process may claim the semaphore.

Acquiring Exclusive Access

To acquire exclusive access to a resource, a process may use the `_os_sema_p()` C binding to reserve the semaphore. If the semaphore is already busy, the process is suspended and placed at the end of the semaphore's wait queue.

Releasing Exclusive Access

To release exclusive access to a resource, a process may use the `_os_sema_v()` C binding to release the semaphore. When the owner process releases the semaphore, the first process in the semaphore's queue is activated and retries the reserve operation on the semaphore.

Semaphores use the following data structure:

```
/* Semaphore structure definition */
typedef struct semaphore  {
    u_int32   s_value,        /* semaphore value (free/busy status) */
              s_lock;         /* semaphore structure lock (use count)*/
    Pr_desc   s_qnext,        /* wait queue for process descriptors */
              s_qprev;        /* wait queue for process descriptors */
    u_int32   s_length,       /* current length of wait queue */
              s_owner,        /* current owner of semaphore */
              s_reserved[6];  /* reserved space */
} semaphore, *Semaphore;
```

Pipes

An OS-9 ***pipe*** is a first-in first-out (FIFO) buffer that allows 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 the OS-9 file manager supporting interprocess communication through pipes. `Pipeman` is a re-entrant subroutine package called for I/O service requests to a device named `/pipe`. Although no physical device is used in pipe communications, you must specify a driver in the pipe descriptor module. The null driver (a driver doing nothing) is usually used, but only gets called by pipeman for GetStat and SetStat calls.

A pipe may contain up to 90 bytes unless a different buffer size was declared in the device descriptor. Typically, a pipe is used as a one-way data path between two processes: one writing and one reading. The reader waits for the data to become available, and the writer waits for the buffer to empty. However, any number of processes can access the same pipe simultaneously; `pipeman` coordinates these processes. A process can even arrange for a single pipe to have data sent to itself. You could use this to simplify type conversions by printing data into the pipe and reading it back using a different format.

You can use pipes much like signals to coordinate processes, but with these advantages:

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

OS-9 supports both named and unnamed (anonymous) pipes. The shell uses unnamed pipes extensively to construct program ***pipelines***, but user programs may use them as well. You may only open a particular unnamed pipes once. Independent processes may communicate

through them only if the pipeline was constructed by a common parent to the processes. Do this by making each process inherit the pipe path as one of its standard I/O paths.

Named and unnamed pipes function nearly identically. The main difference is several independent processes may open a named pipe, which simplifies pipeline construction. The following sections note other specific differences.

Operations on Pipes

Creating Pipes

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

For More Information

Refer to Appendix D: OS-9 for 68K System Calls for more information about `I$Create`.

You may create pipes 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 cannot 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 creation. You may specify access permissions; they are handled similarly to RBF.

The size of the default FIFO buffer associated with a pipe is specified in the pipe device descriptor. You may override this when you create 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 90-byte FIFO buffer inside the path descriptor is used.

Opening Pipes

When accessing unnamed pipes, I$Open, like I$Create, opens a new anonymous pipe file. When accessing named pipes, I$Open searches for the specified name through a linked list of named pipes associated with a particular pipe device. If I$Open finds the pipe, the path number returned refers to the same physical path allocated when the pipe was created. Internally, this is similar to the I$Dup system call.

Opening an unnamed pipe is simple, but sharing the pipe with another process is more complex. If a new path to /pipe is opened for the second process, the new path is independent of the old one.

The only way for more than one process to share the same unnamed pipe is by inheriting the standard I/O paths through the F$Fork call. As an example, the following outline describes a method in pseudo-code the shell might use to construct a pipeline for the command dir -u ! qsort. It assumes paths 0 and 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 several processes may communicate through the same named pipe without having to inherit it from a common parent process. For example, you can approximate the above steps with the following command:

```
dir -u >/pipe/temp & qsort </pipe/temp
```

Note

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

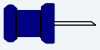
Read/ReadIn

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

For More Information

Refer to Appendix D: OS-9 for 68K System Calls for more information about `I$Read` and `I$ReadLn`.

The end-of-file is recognized when the pipe is empty and the number of processes waiting to read the pipe is equal to the number of users on the pipe. If any data was read before end-of-file was reached, an end-of-file error is not returned. However, the byte count returned is the number of bytes actually transferred, which is less than the number requested.



Note

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

The I$Write and I$WritLn system calls work in almost the same way as I$Read and I$ReadLn. A pipe error (E$Write) is returned when all the processes with a full unnamed pipe open are attempting to write to the pipe. Each process attempting to write to the pipe receives the error, and the pipe remains full.

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

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

Pipeman supports a wide range of status codes to allow you to insert a pipe 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.

The following are the GetStat status codes:

Table 4-9 GetStat Status Codes

Name	Description
SS_Opt	Read the 128-byte option section of the path descriptor. You can use it to get the path type, data buffer size, and name of pipe.
SS_Ready	Test whether data is ready. Returns the number of bytes in the buffer.
SS_Size	Return the size of the pipe buffer.

Table 4-9 GetStat Status Codes (continued)

Name	Description
SS_EOF	Test for end-of-file.
SS_FD	Return a pseudo-file descriptor image.

Other codes are passed to the device driver.

The following are the SetStat status codes:

Table 4-10 SetStat Status Codes

Name	Description
SS_Attr	Change the pipe file's attributes.
SS_Break	Force disconnection.
SS_FD	Do nothing, but return without error.
SS_Opt	Do nothing, but return without error.
SS_Relea	Release the device from the SS_SSig processing before data becomes available.
SS_Size	Reset the pipe buffer if the specified size is zero. Otherwise it has no effect, but returns without error.
SS_SSig	Send a signal when the data becomes available.

Other codes are passed to the device driver.

The I$MakDir and I$ChgDir service requests are illegal service routines on pipes. They 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, which normally read an RBF directory file sequentially, to work with pipes as well.

The head of a linked list of named pipes is in the static storage of the pipe device driver (usually the `null` driver). If several pipe descriptors with different default pipe buffer sizes are on a system, the I/O system notices the same file manager, device driver, and port address (usually zero) are being used. It does not allocate new static storage for each pipe device and all named pipes are on the same list.

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

Data Modules

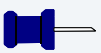
OS-9 data modules allow multiple processes to share a data area and to transfer data among themselves. A ***data module*** must have a valid CRC and module header to be loaded. A data module can be non-re-entrant; it can modify itself and be modified by several processes.

OS-9 does not restrict the content, organization, or use of the data area in a data module. The processes using the data module determine these considerations.

OS-9 does not synchronize processes using a data module. Consequently, thoughtful programming usually involving events, signals or semaphores is required to allow several processes to update a shared data module simultaneously.

Creating Data Modules

The `F$DatMod` system call creates a data module with a specified set of attributes, data area size, and module name. The data area is cleared automatically. The data module is created with a CRC of zero and entered into the system module directory.



Note

It is essential the data module's header and name string not be modified to prevent the module from becoming unknown to the system.

The Microware C compiler provides several C calls to create and use data modules directly. These include the `_mkdata_module()` call, which is specific to data modules, and the `modlink()`, `modload()`, `munlink()`, `munload()`, `_os_datmod()`, `_os_link()`, `_os_unlink()`, `_os_setcrc()`, and `_setcrc()` facilities which apply to all OS-9 modules.

For More Information

For more information about these C calls, refer to the library section of the ***Using Ultra C*** manual.

Link Count

Like all OS-9 modules, data modules have an associated link count. The link count is a counter of how many processes are currently linked to the module. Generally, the module is taken out of memory when this count reaches zero. If you want the module to remain in memory when the link count is zero, create the module with the sticky bit set in its attribute byte.

Saving to Disk

If a data module is saved to disk, you can use the `dump` utility to examine the module's format and contents. You can save a data module to disk using the `save` utility or by writing the module image into a file.

The module CRC of a data module is not valid at creation time, and a valid CRC becomes valid once the data module is modified. A saved data module cannot be reloaded into memory unless you:

- Use the `F$SetCRC` system call or `_setcrc()` C library call before writing the module to disk, or
- Use the `fixmod` utility after the module has been written to disk.

For More Information

For more information about:

- `F$SetCRC` refer to Appendix D: OS-9 for 68K System Calls.
- `dump`, `save`, or `fixmod` refer to the ***OS-9 Utilities Reference Manual***.
- `_setcrc()` refer to the ***Using Ultra C*** manual.

Chapter 5: User Trap Handlers

This chapter explains how to install and execute trap handlers, and provides an example of trap handler coding. It includes the following topics:

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. You may use the remaining fifteen 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:

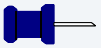
Table 5-1 Reserved Traps

Trap	Description
13	`cio/csl` is automatically called for any C program. `cio` and `csl` use the same trap. C programs are compiled to use one or the other but not both. Selection depends on the libraries linked into the program at link time. UCC libraries use `csl` while the original C compiler (now OC) used libraries that trapped to `cio`. See your C compiler manual for additional information.
15	`Math` is called for floating point math, extended integer math, and/or type conversion. It is also used for programs using transcendental and/or extended mathematical functions.

For More Information

For further information about the math module, refer to Chapter 6: The Math Module.

A ***user trap handler*** is an OS-9 module usually containing a set of related subroutines. Any user program may dynamically link to the user trap handler and call it at execution time.



Note

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 calling trap handlers execute slightly slower than linked programs performing 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 for use by a number of programs.

Trap handlers are similar to normal OS-9 program modules, except trap handlers have three execution entry points:

- A trap execution entry point
- A trap initialization entry point
- A trap termination entry point

Trap handler modules are of module type `TrapLib` and module language `Objct`.

The trap module routines usually execute as though they were called with a `jsr` instruction, except for minor stack differences. Any system calls or other operations the calling module could perform are usable in the trap module.

You can write a trap handler module running in system state. This is rarely advisable, but sometimes necessary.

For More Information

Refer to the **System Call Overview** section in Chapter 2: The Kernel, for more information about the uses of system-state.

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)
- Executes the trap module's initialization routine

For More Information

Refer to Appendix D: OS-9 for 68K System Calls for more information about `F$TLink`.

Typically, the initialization routine has very little to do. You could use it to:

- Open files
- Link to additional trap or data modules
- Perform other startup activities

It is called only once per trap handler in any given program.

A trap module used by a program is usually installed as part of the program's initialization code. At initialization, a particular trap number (1-15) is specified that refers to the trap module. The program calls functions in the trap module by using the 68000 `trap` instruction corresponding to the trap number specified. This is followed by a function word 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
- `tcall` for the other user traps

They work like built-in macros, generating code as illustrated in the following section.

OS9 and tcall: Equivalent Assembly Language Syntax

The following shows equivalent assembly language syntax for `os9` and `tcall`:

Table 5-2 OS-9/tcall Equivalencies

Mnemonic	Code Generation
`OS9 F$TLink`	`trap 0` `dc.w F$TLink`
`tcall T$Math,T$DMul`	`trap T$Math` `dc.w T$DMul`

From user programs, you can 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). The program aborts if this offset is zero. Otherwise, OS-9 passes control to the exception routine. At this point, the trap handler can be installed, and the first `tcall` reissued. The second example in this chapter shows how to do this.

Calling a Trap Handler

The actual details of building and using a trap handler are best explained by means of a simple, complete example.

Example One

The following program (`TrapTst`) uses trap vector 5. It installs the trap handler and then calls it twice.

```
            nam     TrapTst1
            ttl     example one - link and call trap handler
            use     /dd/defs/oskdefs.d
Edition     equ     1
Typ_Lang    equ     (Prgrm<<8)+Objct
Attr_Rev    equ     (ReEnt<<8)+0
            psect   traptst,Typ_Lang,Attr_Rev,Edition,1024,Test

TrapNum     equ     5               trap number to use
TrapName    dc.b    "trap",0        name of trap handler

*******************************
* Main program entry point

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 you could modify the preceding program to install the trap handler in an exception routine when the first `tcall` is executed. You might do this for a trap handler that may not be used at all by a program, depending on circumstances.

This example does not initialize the trap handler before using it, but is otherwise identical to **Example One**. It provides a `LinkTrap` subroutine to automatically install the trap handler when it is first used. Refer to the **Trace of Example Two Using the Example Trap Handler** later in this chapter for more information.

```
            nam     TrapTst2
            ttl     example two - call trap handler
            use     /dd/defs/oskdefs.d
Edition     equ     1
Typ_Lang    equ     (Prgrm<<8)+Objct
Attr_Rev    equ     (ReEnt<<8)+0
            psect   traptst,Typ_Lang,Attr_Rev,Edition,1024,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

*********************************
* Subroutine LinkTrap
* Installs trap handler and then executes first trap call.
* Note: Error checking is minimized to keep example simple.
*
* Passed:  d0-d7 = caller's registers
*          a0-a5 = caller's registers
*          (a6)  = trap handler static storage pointer
*          (a7)  = trap init/entry stack frame
*
* Returns: trap installed, backs up PC to execute "tcall" instruction
*
* The 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 following `makefile` makes the example trap handler and test programs:

```
# makefile - Used to make the example trap handler and test programs.

RDIR   = RELS
TRAP   = trap
TEST1  = traptst1
TEST2  = traptst2

# Dependencies for making the entire trap example.

trap.example: $(TRAP) $(TEST1) $(TEST2)
    touch trap.example

# Dependencies for making the trap handler.

$(TRAP): $(TRAP).r
    l68 -g $(RDIR)/$(TRAP).r -l=/dd/lib/sys.l -o=$(TRAP)

# Dependencies for making the traptst1 test program.

$(TEST1): $(TEST1).r
    l68 -g $(RDIR)/$(TEST1).r -l=/dd/lib/sys.l -o=$(TEST1)

# Dependencies for making the traptst2 test program.

$(TEST2): $(TEST2).r
    l68 -g $(RDIR)/$(TEST2).r -l=/dd/lib/sys.l -o=$(TEST2)
```

The trap handler itself is listed here. It is 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 handler module
          use     /dd/defs/oskdefs.d
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

**************************************************
* TrapInit: Trap handler initialization entry point.
*
* Passed:  d0.w  = User Trap number (1-15)
*          d1.l  = (optional) additional static storage
*          d2-d7 = caller's registers at the time of the trap
*          (a0)  = trap handler module name pointer
```

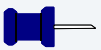
```
*          (a1)  = trap handler execution entry point
*          (a2)  = trap module pointer
*          a3-a5 = caller's registers (parameters required by handler)
*          (a6)  = trap handler static storage pointer
*          (a7)  = trap init stack frame pointer
*
* Returns: (a0)  = updated trap handler name pointer
*          (a1)  = trap handler execution entry point
*          (a2)  = trap module pointer
*          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            restore user's a6 register
          addq.l  #8,a7              take other stuff off the stack
          rts                        return to caller

****************************************
* TrapEnt: User trap handler entry point.
*
* Passed:  d0-d7 = caller's registers
*          a0-a5 = caller's registers
*          (a6)  = trap handler's static storage pointer
*          (a7)  = trap entry stack frame pointer
*
* Returns: cc    = carry set, d1.w=error code if error
*          Other values returned are dependent on the trap handler
*
* The stack looks like this:
*          .-------------------------.
*       +8 |   caller's return PC    |
*          >-------------------------<
*       +6 |  vector #  |
*          >------------<
*       +4 | func code  |
*          >-------------------------<
*          |  caller's a6 register   |
*    (a7)-> -------------------------

          org     0                  stack offset definitions
S.d0      do.l    1                  caller's d0 reg
S.d1      do.l    1                  caller's d1 reg
S.a0      do.l    1                  caller's a0 reg
S.a6      do.l    1                  caller's a6 reg
S.func    do.w    1                  trap function code
S.vect    do.w    1                  vector number
S.pc      do.l    1                  return pc
```

```
TrapEnt:  movem.l d0-d1/a0,-(a7)      save registers
          move.w  S.func(a7),d0       get function code
          cmp.w   #1,d0               is function in range?
          bhi.s   FuncErr             abort if not
          beq.s   Trap10              branch if function code #1
          lea     String1(pc),a0      get first string ptr
          bra.s   Trap20              continue
Trap10    lea     String2(pc),a0      get second string ptr
Trap20    moveq   #1,d0               standard output path
          moveq   #80,d1              maximum bytes to write
          os9     I$WritLn            output the string
          bcs.s   Abort               abort if error
Trap90    movem.l (a7)+,d0-d1/a0/a6-a7 restore regs
          rts                         return to user

FuncErr   move.w  #1<<8+99,d2         abort (return error 001:099)
Abort     move.w  d1,S.d1+2(a7)       put error code in d1.w
          ori     #Carry,ccr          set carry
          bra.s   Trap90              exit

String1   dc.b    "Microware Systems Corporation",C$CR,0
String2   dc.b    "    Quality keeps us #1",C$CR,0

*********************************************
* TrapTerm: Trap handler terminate entry point.
*
* As of this release (OS-9 V2.4) 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 to watch the OS-9 user debugger trace through the execution of Example Two (using the example trap handler). User trap handlers look like subroutines to the debugger, so it is possible to trace through them. The output should appear something like this:

```
(beginning of second example program)
Test                >4E450000         trap #5,0
```



Note

Because the trap handler has not been linked as in **Example One**, control jumps to the subroutine `LinkTrap`:

```
LinkTrap            >508F             addq.l #8,a7
LinkTrap+0x2        >48E7C0E0         movem.l d0-d1/a0-a2,-(a7)
LinkTrap+0x6        >7005             moveq.l #5,d0
LinkTrap+0x8        >7200             moveq.l #0,d1
LinkTrap+0xA        >41FAFFDC         lea.l bname+0xA(pc),a0
LinkTrap+0xE        >4E400021         os9 F$TLink
```

Note

Control switches to the subroutine `TrapInit` and then returns to `LinkTrap`:

```
trap:btext+0x50     >4CD74000         movem.l (a7),a6
trap:btext+0x54     >508F             addq.l #8,a7
trap:btext+0x56     >4E75             rts
LinkTrap+0x12       >65E8             bcs.b Test+0xE
LinkTrap+0x14       >4CDF0703         movem.l (a7)+,d0-d1/a0-a2
LinkTrap+0x18       >5997             subq.l #4,(a7)
LinkTrap+0x1A       >4E75             rts
```

Note

Control now returns to the main program to re-execute the `tcall` instruction.

```
Test                >4E450000         trap #5,0
trap:TrapEnt        >48E7C080         movem.l d0-d1/a0,-(a7)
trap:TrapEnt+0x4    >302F0010         move.w 16(a7),d0
trap:TrapEnt+0x8    >B07C0001         cmp.w #1,d0
trap:TrapEnt+0xC    >621C             bhi.b trap:TrapEnt+0x2A
trap:TrapEnt+0xE    >6706             beq.b trap:TrapEnt+0x16
trap:TrapEnt+0x10   >41FA0026         lea.l trap:TrapEnt+0x38(pc),a0
trap:TrapEnt+0x14   >6004             bra.b trap:TrapEnt+0x1A
trap:TrapEnt+0x1A   >7001             moveq.l #1,d0
trap:TrapEnt+0x1C   >7250             moveq.l #80,d1
trap:TrapEnt+0x1E   >4E40008C         os9 I$WritLn
Microware Systems Corporation
trap:TrapEnt+0x22   >650A             bcs.b trap:TrapEnt+0x2E
trap:TrapEnt+0x24   >4CDFC103         movem.l (a7)+,d0-d1/a0/a6-a7
trap:TrapEnt+0x28   >4E75             rts
Test+0x4            >6508             bcs.b Test+0xE
Test+0x6            >4E450001         trap #5,0x1
trap:TrapEnt        >48E7C080         movem.l d0-d1/a0,-(a7)
trap:TrapEnt+0x4    >302F0010         move.w 16(a7),d0
trap:TrapEnt+0x8    >B07C0001         cmp.w #1,d0
trap:TrapEnt+0xC    >621C             bhi.b trap:TrapEnt+0x2A
trap:TrapEnt+0xE    >6706             beq.b trap:TrapEnt+0x16->
trap:TrapEnt+0x16   >41FA003F         lea.l trap:TrapEnt+0x57(pc),a0
trap:TrapEnt+0x1A   >7001             moveq.l #1,d0
trap:TrapEnt+0x1C   >7250             moveq.l #80,d1
trap:TrapEnt+0x1E   >4E40008C         os9 I$WritLn
    Quality keeps us #1
trap:TrapEnt+0x22   >650A             bcs.b trap:TrapEnt+0x2E
trap:TrapEnt+0x24   >4CDFC103         movem.l (a7)+,d0-d1/a0/a6-a7
trap:TrapEnt+0x28   >4E75             rts
Test+0xA            >6502             bcs.b Test+0xE
Test+0xC            >7200             moveq.l #0,d1
Test+0xE            >4E400006         os9 F$Exit
```

Chapter 6: The Math Module

This chapter discusses math module functions, and lists descriptions of the assembler calls you can use with the math module. It includes the following topics:

- **Introduction**
- Floating Point Co-processor Emulation Modules
- **Math Trap Handler**

Introduction

OS-9 contains two module types supporting floating point math calculations:

Floating Point Co-processor Emulation Modules

For systems lacking an MC68881 or MC68882 floating point co-processor, the co-processor emulation modules `fpu` and `fpsp040` provide a seamless software solution for code compiled to use the MC68882 instruction set. This includes all code the Ultra C compiler generates, as well as code generated using the Version 3.2 compiler with the `-k=2f` option.

Math Trap Handler

The math trap handlers, `math` and `math881`, are provided for backward compatibility with code generated using the Version 3.2 compiler with the `-x` option.

Floating Point Co-processor Emulation Modules

The floating point instructions supported by the MC68881 and MC68882 co-processors cause a floating point exception when executed on a system without a co-processor. This results in the process aborting with the error:

```
Error #000:111(E$1111)
A "1111" instruction exception occurred
```

To prevent this error and allow proper emulation of the faulted floating point instruction, OS-9 provides the co-processor emulation module `fpu` (and `fpsp` for the MC68040).

The co-processor emulation module provides an exception handler that traps the floating point exception, decodes the faulted instruction, and emulates (in the software) the functionality of the floating point instruction. OS-9 for 68K V3.0 contains two such emulation modules:

Table 6-1 Emulation Modules

Emulation Module	Description
`fpu`	The `fpu` module is compatible with all OS-9 for 68K Version 3.0 systems. `fpu` supports the subset of the MC68882 programming model used by Microware's V3.2 compiler and Ultra C compiler. See ***Using Ultra C*** for additional information on the instructions supported by `fpu`.

Table 6-1 Emulation Modules (continued)

Emulation Module	Description
`fpsp`	The MC68040 contains a built-in subset of the MC68882 programming model. The `fpsp` module provides the remainder of the MC68882 functionality. `fpsp`, with the MC68040, provides full support of the MC68882 instruction set. The `fpsp` module is only supported for the MC68040.

Note

While `fpu` is supported on the MC68040, Microware does not recommend its use. Instead, use the MC68040-specific `fpsp` module, as it provides increased speed, precision, and functionality.

Installing Co-processor Emulation Modules

The modules `fpu` and `fpsp` are provided in the following files:

```
fpu  MWOS/OS9/CMDS/68000/BOOTOBJS/fpu
fpsp MWOS/OS9/CMDS/68020/BOOTOBJS/fpsp040
```

These modules are OS-9 extension modules and must be available during bootup.

To install either module, add the executable module to your bootfile for disk based systems or include it in ROM for diskless systems. You also need to add the name of the module (`fpu` or `fpsp`) to your system's `Init` module's extensions list.

During bootup, the kernel calls each extensions module listed in the `Init` module's extensions list. When the co-processor emulation module is called in this fashion, it installs its floating point exception handler on OS-9 and initializes the data structures necessary for floating point emulation. This is transparent to the user.

The following are the errors you may get from the emulation routines during installation:

`E$KwnMod`	An emulation module has already been installed (`fpu` and `fpsp`).
`E$BadId`	For OS-9 for 68K V3.0, `fpu` returns this error if it was invoked by a module other than the kernel's bootup routine.
`E$BadRev`	`fpu` returns this error if you make an attempt to install `fpu` on an incompatible OS-9 revision/hardware platform.

Math Trap Handler

Microware's Version 3.2 compiler provided a means to access math functions from within the `math` shared trap library.

While OS-9 for 68K V3.0 and its utilities no longer require the `math` module, it is provided for backward compatibility with third party executables and relocatable libraries compiled with the V3.2 compiler. An attempt to run such code on a system without the `math` trap handler results in the error:

```
**** Can't install trap handler ****
 **** math ****
Error #000:216
```

The `math` module is provided in two forms, depending on the availability of an MC68881/MC68882 co-processor in the target system. While both modules are named `math`, they are differentiated by the names of the files in which they are provided.

`MWOS/OS9/68000/CMDS/math`
: provides software support for the routines in the math library. While this module is compatible on systems with or without floating point co-processors, Microware suggests you use the faster `math881` module on systems with co-processors.

`MWOS/OS9/68020/CMDS/math881`
: provides a version of the math library routines that takes advantage of the MC68882 instruction set. While this is the preferred module for use on a system with a co-processor, a floating point exception occurs if used on a system with no co-processor. (Refer to Floating Point Co-processor Emulation Modules earlier in this chapter.)

To install either module, simply load it into memory.

Note

As the `math` module may be discontinued in a future release, Microware suggests you replace all such executable and relocatable code with code compiled with Ultra C.

Chapter 7: OS-9 File System

This chapter explains OS-9's disk file organization, raw physical I/O on RBF devices, record locking, and file security. It includes the following topics:

- **Disk File Organization**
- **Raw Physical I/O on RBF Devices**
- **Record Locking**
- **File Security**

Disk File Organization

RBF supports a tree-structured file system. The physical disk organization is designed for:

- Efficient use of disk space
- Resistance to accidental damage
- Fast file access

The system also has the advantage of relative simplicity.

Basic Disk Organization

RBF supports logical sector sizes in integral binary multiples from 256 to 32768 bytes. If you use a disk system that cannot directly support the logical sector size (for example, 256 byte logical sectors on a 512-byte physical sector disk), the driver module must divide or combine sectors as required to simulate the required logical size.

Many disks are physically addressed by:

- Track number
- Surface number
- Sector number

To eliminate hardware dependencies, OS-9 uses a ***logical sector number*** (LSN) to identify each sector without regard to track and surface numbering.

The disk driver module or the disk controller is responsible for mapping logical sector numbers to track/surface/sector addresses. OS-9's file system uses LSNs from 0 to (n-1), where ***n*** is the total number of sectors on the drive.

Note

All sector addresses covered in this section refer to LSNs.

The `format` utility initializes the file system on blank or recycled media by creating the track/surface/sector structure. `format` also tests the media for bad sectors and automatically excludes them from the file system.

For More Information

Refer to ***Utilities Reference*** for information about using `format`.

Every OS-9 disk has the same basic structure:

Identification Sector	located in logical sector zero (LSN 0). It contains a description of the physical and logical format of the storage volume (disk media).
Allocation Map	usually beginning in logical sector one (LSN 1). This indicates which disk sectors are free for use in new or expanded files.
Root Directory	begins immediately after the disk allocation map.

Identification Sector

LSN zero always contains the identification sector (see **Table 7-1**). It describes the physical format of the disk, the size of the allocation map, and the location of the root directory. It also contains the volume name,

date and time of creation, and additional information. If the disk is a bootable system disk, it also has the starting LSN and size of the `OS9Boot` file.

Table 7-1 Identification Sector Description

Addr	Size	Name	Description
$00	3	DD_TOT	Total number of sectors on media
$03	1	DD_TKS	Track size in sectors
$04	2	DD_MAP	Number of bytes in allocation map
$06	2	DD_BIT	Number of sectors/bit (cluster size)
$08	3	DD_DIR	LSN of root directory file descriptor
$0B	2	DD_OWN	Owner ID
$0D	1	DD_ATT	Attributes
$0E	2	DD_DSK	Disk ID
$10	1	DD_FMT	Disk Format; density/sides Bit 0: 0 = single side 1 = double side Bit 1: 0 = single density (FM) 1 = double density (MFM) Bit 2: 1 = double track (96 TPI/135 TPI) Bit 3: 1 = quad track density (192 TPI)

Table 7-1 Identification Sector Description (continued)

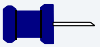
Addr	Size	Name	Description
$11	2	DD_SPT	Sectors/track (two byte value DD_TKS)
$13	2	DD_RES	Reserved for future use
$15	3	DD_BT	System bootstrap LSN. 0 = no boot present
$18	2	DD_BSZ	Size of system bootstrap
$1A	5	DD_DAT	Creation date
$1F	32	DD_NAM	Volume name
$3F	32	DD_OPT	Path descriptor options
$5F	1		Reserved
$60	4	DD_SYNC	Media integrity code
$64	4	DD_MapLSN	Bitmap starting sector number (0=LSN 1)
$68	2	DD_LSNSize	Media logical sector size (0=256)
$6A	2	DD_VersID	Sector 0 Version ID

Allocation Map

The allocation map shows which sectors are:

- Allocated to files
- Free for future use

DD_MapLSN specifies the allocation map start address, which is usually 1. If this field is 0, assume an address of 1. The size of the map varies according to how many bits are needed. Each bit in the allocation map represents a cluster on the disk. If a bit is set, the cluster is considered to be in use, defective, or non-existent. DD_MAP (see **Table 7-1**) specifies the actual number of bytes used in the map.



Note

The DD_Bit variable specifies the number of sectors per cluster. The number of sectors per cluster is always an integral power of two.

The format utility sets the size of the allocation map depending on the size and number of sectors per cluster. You can select the number of sectors per cluster on the command line when using format.

Root Directory

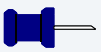
The root directory file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as /d1). Usually, it immediately follows the allocation map. The location of the root directory file descriptor is specified in DD_DIR (see **Table 7-1**).

Basic File Structure

OS-9 uses a multiple-contiguous-segment type of file structure. Segments are physically contiguous sectors that store the file's data. If all the data cannot be stored in a single segment, additional segments are allocated to the file. This may occur if a file is expanded after creation or if a sufficient number of contiguous free sectors is not available.

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

All files have a sector called a file descriptor sector, or FD. FD contains a list of the data segments with their starting LSNs and sizes. This is also where information such as file attributes, owner, and time of last modification is stored. Only the system uses this sector; you cannot directly access it. **Table 7-2** describes the contents of a file descriptor.



Note

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`.

Table 7-2 FIle Descriptor Content Description

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).

Table 7-2 FIle Descriptor Content Description (continued)

Offset	Size	Name	Description
$0D	3	FD_CREAT	Date Created: Y M D
$10	240	FD_SEG	Segment List: see below.

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 indicates public execute, bit 4 indicates public write, and so forth.

The date last modified (FD_DAT) changes when a file is opened in write or update mode. This is useful for making date-dependent backups.

The segment list (FD_SEG) consists of a series of five-byte entries, continuing until the end of the logical sector. For 256-byte sectors, this results in 48 entries. These entries 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 specifying the beginning of the block and a two-byte block size (in sectors). Unused segments must be zero.

The RBF file manager maintains the file pointer and logical end-of-file used by application software and converts them to the logical disk sector number using the data in the segment list.

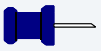
Note

You do not have to be concerned with physical sectors. 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***. It 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, you should set this field to a relatively high value, and vice versa.

When a file is created, it 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 allocation of additional sectors 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 before attempting to add a new segment.

If not all of the allocated sectors are used when the file is closed, the segment is truncated and any unused sectors are deallocated in the bitmap. This strategy does not work well for random-access databases that expand frequently by only a few records. The segment list is rapidly filled 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 is not truncated. To be effective, all programs dealing with the file in write or update mode must ensure they do not close the file while at end-of-file, or the file loses any excess space it may have. The easiest way to ensure this is to do a `seek(0)` before closing the file. This method was chosen because random access files are frequently somewhere other 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 file's file descriptor address:

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

When a directory file is created, two entries are automatically created: the dot (`.`) and double dot (`..`) directory entries. These specify the directory and its parent directory, respectively.

Raw Physical I/O on RBF Devices

You can open an entire disk as one logical file. This allows you to access any byte(s) or sector(s) 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.

To open a device for physical I/O, append an at (@) character to the device name. For example, you can open the device `/d2` 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 the logical sector size of the media. You can find the logical sector size in the `PD_SctSiz` field of the path descriptor (if 0, assume a value of 256 bytes). For example, to read sector 3 on 1024-byte logical media, perform a seek to address 3072 (1024 * 3), followed by a read system call requesting 1024 bytes.

If the number of sectors per track 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

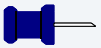
Use extreme care with the special @ file in update mode. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The @ file is considered different from any other file, and therefore only conforms to record lockouts with other users of the @ file.

Improper physical I/O operations can corrupt the file system. Take great care when writing to a raw device. Physical I/O calls also bypass the file security system. For this reason, only super-users are allowed to open

the raw device for write permit. Non-super-users can only 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 refers to preserving the integrity of files that more than one user or process can access. OS-9 record locking is designed to be as invisible as possible to application programs.



Note

Most programs may be written without special concern for multi-user activity.

Record locking involves:

- Recognizing when a process is trying to read a record that another process may be modifying
- Deferring the read request until the record is safe

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

Record Locking and Unlocking

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

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

When a file is opened in update mode, ***any*** read causes 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 opened in read or execute modes does not cause record locking to occur because records cannot be updated in these two modes.

A subtle but nasty problem exists for programs that interrogate a database 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; you can avoid it by careful programming.

Note

Only one portion of a file may be locked out at a time. If an application requires more than one record to be locked out, multiple paths to the same file may be opened with each path having its own record locked out. RBF notices the same process owns both paths and keeps them from locking each other out. Alternatively, the entire file may be locked out, the records updated, and the file released.

Non-Sharable Files

You may use file locking when an entire file is considered unsafe for use by more than one user. On rare occasions, you need to create a ***non-sharable file***. A non-sharable file can never be accessed by more than one process at a time.

To make a file non-sharable, set the single user (S) bit in the file's attribute byte. You can set the S bit when you create the file, or later using the `attr` utility. If the single-user bit is set, only one process may open the file at a time. If another process attempts to open the file, error (#253) is returned.

For More Information

Refer to ***Utilities Reference*** for more information about `attr`.

More commonly, a file needs to be non-sharable only while a specific program is executing. To do this, open the file with the single-user bit set in the access mode parameter.

For example, if a file is opened as a non-sharable file, when it is being sorted it is treated as though it had a single-user attribute. If the file was already opened by another process, an error (#253) is returned.

A necessary quirk of non-sharable files is they may be:

- Duplicated using the `I$Dup` system call
- Inherited

A non-sharable file could therefore actually become accessible to more than one process at a time. Non-sharable only means the file may be opened once.

Note

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 you read or write data at the end of file. You keep the end of file locked until you perform a read or write that is not at the end of the file. EOF lock is the only time a write call automatically locks out of any part of the file. This avoids problems occurring when two users try to simultaneously extend a file.

An extremely useful side effect occurs when a program creates a file for sequential output. When the file is created, EOF lock is gained, and no other process can pass the writer in processing the file.

For example, if you redirect an assembly listing 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 always keeps 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 simultaneously. If each process gets one area (locking out the other process), both processes are 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. At least one process must release its control over a requested segment for any to proceed.

Record Locking Details for I/O Functions

The following lists record locking details for I/O functions:

Table 7-3 Record Locking Details for I/O Functions

Function	Description
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 do not cause records to be locked out, and they generally help the system to run faster. If shared files are routinely opened for update on a multi-user system, you can become hopelessly record-locked for extended periods of time. Use the special @ file 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 only conforms to record lockouts with other users of the @ file.

Table 7-3 Record Locking Details for I/O Functions (continued)

Function	Description
Read/ ReadLine	Read and ReadLine cause lock out of records 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 you make a ReadLine call for 256 bytes, exactly 256 bytes are locked out, regardless of how many bytes are actually read before a carriage return is encountered. EOF lock occurs if the bytes requested include the current end-of-file. A record remains locked until any of the following occur: • Another read is performed • A write is performed • The file is closed • A record lock SetStat is issued Releasing a record does not normally release EOF lock. Any read or write of zero bytes releases any record lock, EOF lock, or file lock.
Write/ WriteLine	Write calls always release any record that is 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 gains EOF lock.
Seek	Seek does not effect record locking.

Table 7-3 Record Locking Details for I/O Functions (continued)

Function	Description
SetStat	There are two SetStat codes to deal with record locking: • `SS_Lock` locks or releases part of a file • `SS_Ticks` sets the length of time a program waits for a locked record See the `I$SetStt` entry in Appendix D: OS-9 for 68K System Calls for a description of the codes.

File Security

Each file has a group/user ID identifying the file's owner. These are copied from the current process descriptor when the file is created. Usually, a file's owner ID is not changed.

An attribute byte is also specified when a file is created. The file's attribute byte tells RBF in which modes the 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 to 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 with the same group ID as the file's creator. Public means any user with a different group ID.

When 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.

Any ***super-user*** (a user with group ID of zero) may access any file in the system. Files owned by the super-user cannot be accessed by users of any other group unless specific access permissions are set. Files containing modules owned by the super-user must also be owned by the super-user. If not, the modules contained within the file are not loaded.

Note

The system manager should exercise caution 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 residing in the password file is permitted two bytes for the group ID and two bytes for the user ID. RBF only reads the low order byte of both the group and user ID. Consequently, a user with the ID of 256.512 is mistaken for the super user by RBF.

Appendix A: Example Code

This appendix contains example code you can use as a guide when creating your own modules. It provides examples of RBF, SCF, SBF, and pipe device descriptors. It includes the following topics:

The Init Module

The following is an example of the `Init` module:

```
Microware OS-9/68020 Resident Macro Assembler V2.9  93/10/12  11:15  Page     1
 ../../../SRC/SYSMODS/INIT/init.a
Init: OS-9 Configuration Module - LRCChip.d - Local Resource Controller
definitions
00001                       nam      Init: OS-9 Configuration Module
00002
00003 * Copyright 1983, 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1991, 1992, 1993
00004 * 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 than
00011 * the licensee is strictly prohibited.
00012
00078 *
00079  00000029 Edition    equ      41             current edition number
00080
00081  00000c00 Typ_Lang   set      (Systm<<8)+0
00082  00008000 Attr_Rev   set      (ReEnt<<8)+0
00083
00084                       psect    init,Typ_Lang,Attr_Rev,Edition,0,0
00085
00086 * Configuration constants (default; changable in "systype.d" file)
00087 *
00088 * Constants using VALUES (such as CPUTyp set 68020) may appear anywhere
00089 * in the "systype.d" file.
00090 * Constants using LABELS (such as Compat set ZapMem) MUST appear OUTSIDE
00091 * the CONFIG macro and must be conditionalized so they are
00092 * only invoked when this file (init.a) is being assembled.
00093 * If they are placed inside the CONFIG macro, then the over-ride does not
00094 * take effect.
00095 * If they are placed outside the macro and not conditionalized then
00096 * "illegal external reference" errors result when making other files.
00097 * The label _INITMOD provides the mechanism to ensure the desired
00098 * operations result.
00099 *
00100 * example systype.d setup:
00101 *
00102 * CONFIG macro
00103 *   <body of macro>
00104 *  endm
00105 *  Slice set 10
00106 *  ifdef _INITMOD
00107 * Compat set ZapMem patternize memory
00108 *  endc
00109 *
00110
```

```
00111 * flag reading init module (so local labels can be over-ridden)
00112  00000001 _INITMOD   equ      1              flag reading init module
00113
00114  000109a0 CPUTyp     set      68000         cpu type (68008/68000/68010/etc.)
00115  00000001 Level      set      1               OS-9 Level One
00116  00000003 Vers       set      3               Version 3.0
00117  00000000 Revis      set      0
00118  00000000 Edit       set      0               Edition
00119  00000000 IP_ID      set      0    interprocessor identification code
00120  00000000 Site       set      0    installation site code
00121  00000040 MDirSz     set      64   initial module directory count
00122  00000020 PollSz     set      32   IRQ polling table size (fixed)
00123  00000020 DevCnt     set      32   device table size (fixed)
00124  00000040 Procs      set      64   init process table size (divisible by 64)
00125  00000040 Paths      set      64   initial path table size (divisible by 64)
00126  00000002 Slice      set      2    ticks per time slice
00127  00000080 SysPri     set      128  initial system priority
00128  00000000 MinPty     set      0    initial system minimum executable priority
00129  00000000 MaxAge     set      0    initial system maximum natural age limit
00130  00000000 MaxMem     set      0    top of RAM (unused)
00131  00000020 Events     set      32   initial event table size
00132  00000000 Compat     set      0    version smoothing byte
00133  00000000 Config     set      0    system configuration default
00134  00000400 StackSz    set      1024  IRQ Stack Size in bytes (must be 1k <=
StackSz < 256k)
00135  00000000 ColdRetrys set      0     number of retries for coldstart's "chd"
before failing
00136 * NOTE: for V3.0, NumSigs is unimplemented
00137  00000010 NumSigs    set      16    default queued signal maximum
00138  000005dc PrcDescStack set    1500  default stack size in process descriptor
00139
00140
00141 * Compat flag bit definitions
00142 *
00143 * NOTE: SlowIRQ is obsolete for V3.0
00144 *
00145 *SlowIRQequ 1<<0xxxxxxx1 save all regs during IRQ processing
00146  00000002 NoStop     equ      1<<1  xxxxxx1x don't use 'stop' instruction
00147  00000004 NoGhost    equ     1<<2  xxxxx1xx don't retain Ghost/Sticky memory
modules
00148  00000008 NoBurst    equ      1<<3  xxxx1xxx don't enable 68030 cache burst
mode
00149  00000010 ZapMem     equ      1<<4  xxx1xxxx wipe out memory that is
allocated/freed
00150  00000020 NoClock    equ      1<<5  xx1xxxxx don't start system clock during
coldstart
00151  00000040 SpurIRQ    equ      1<<6  x1xxxxxx ignore spurious IRQs
00152  00000080 PrivAlm    equ     1<<7  1xxxxxxx only alarm creator (process) can
delete alarm.
00153
00154
00155 * Compat2 flag bit definitions
00156 * (these are reserved for cache control considerations)
00157 *
```

```
00158  00000001 ExtC_I     equ      1<<0  xxxxxxx1 external instruction cache is
coherent
00159  00000002 ExtC_D     equ      1<<1  xxxxxx1x external data cache is coherent
00160  00000004 OnC_I      equ      1<<2  xxxxx1xx on-chip instruction cache is
coherent
00161  00000008 OnC_D      equ      1<<3  xxxx1xxx on-chip data cache is coherent
00162  00000010 CBank0_En  equ      1<<4  xxx1xxxx 68349: enable cache/sram bank 0
as cache
00163  00000020 CBank1_En  equ      1<<5  xx1xxxxx 68349: enable cache/sram bank 1
as cache
00164  00000040 CBank2_En  equ      1<<6  x1xxxxxx 68349: enable cache/sram bank 2
as cache
00165  00000080 CBank3_En  equ      1<<7  1xxxxxxx 68349: enable cache/sram bank 3
as cache
00166
00167 * NOTE: DDIO is obsolete for V3.0
00168 *DDIOequ 1<<71xxxxxxx don't disable data caching when in I/O
00169
00170
00171 * Config flag bit definitions
00172 *
00173 * These definitions control various options for the system
00174 *
00175  00000001 NoTblExp   equ      1<<0  xxxxxxxx xxxxxxx1 system table expansion
disabled
00176 * Note - CRCDis is only applicable for "atomic" kernel.
00177  00000004 CRCDis     equ      1<<2  xxxxxxxx xxxxx1xx disable CRC check for
F$VModul
00178  00000008 SysTSDis   equ      1<<3  xxxxxxxx xxxx1xxx disable system-state
time-slicing
00179
00180 * SSM_NoProt is only applicable for Development kernel: SSM  builds
00181 * a "full-system" page table for User-State at cold-start, all user
00182 * processes have access to all memory (ala ATOMIC kernel)
00184
00185 * SSM_SysPT is unimplemented for V3.0
00186  00000020 SSM_SysPT  equ      1<<5  xxxxxxxx xx1xxxxx (SSM) sys-state page
tables
00187
00188
00189 * SSM(MMU) Page Cache Mode definitions
00190 *
00191 * These definitions affect how the SSM(MMU) page tables are setup,
00192 * with respect to USER-state data cache modes.
00193 * Note the options available are dependant upon the caching
00194 * modes supported by the MPU/MMU in use.
00195 *
00196  00000004 CM_WrtProt equ      1<<2  page is write-protected
00197  00000040 CM_CI      equ      1<<6  cache inhibit bit (68851/68030/68040)
00198  00000020 CM_NotSer  equ      1<<5  not-serialized access (for cache
inhibited) (68040)
00199  00000020 CM_CB      equ      1<<5  copy-back (for cache enabled) (68040)
00200
00201  00000004 WritProt   equ      CM_WrtProt     write-protected page (not
usually useful)
```

```
00202
00203 * 68040 definitions
00204 *
00205  00000000 WrtThru    equ     0           cache enabled, write-through (also
68020/030 cache enabled)
00206  00000020 CopyBack   equ      CM_CB        cache enabled, copy-back
00207  00000040 CISer      equ     CM_CI        cache inhibited, serialized access
00208  00000060 CINotSer   equ    CM_CI+CM_NotSer cache inhibited, not-serialized
access
00209
00210
00211                      use     defsfile (any above definitions may be
overridden in defsfile)
00001
00002                      use     <oskdefs.d>
00043
00004                      use     systype.d
00001                      opt     -l
00005
00212
00213 * Memory list definitions
00214           MemType    macro
00215                      dc.w    \1,\2,\3,\4>>4type, priority, access, search
block size
00216                      dc.l    \5,\6low, high limits (where it appears on
local address bus)
00217                      dc.w    \7,0offset to description string (zero if
none), reserved
00218                      dc.l    \8,0,0address translation adjustment (for
DMA, etc.), reserved
00219                      ifne    \#-8  must have exactly eight arguments
00220                      fail    wrong number of arguments to MemType macro
00221                      endc
00222                      endm
00223
00224 * Cache Mode Memory list definitions
00225 *
00226           CacheType  macro
00227                      dc.l    \1,\2   start and end address (+1) of region
00228                      dc.w    0       reserved
00229                      dc.w    \3      cache modes for region
00230                      dc.l    0       reserved
00231                      ifne    \#-3 must have exactly three arguments
00232                      fail    wrong number of arguments to CacheType macro
00233                      endc
00234                      endm
00235
00236 * Alignment macros (for optimization)
00237 *
00238           CPUALIGN   macro
00239                      ifeq    (CPUTyp-68040)
00240 * force LINE alignment
00241                      ifne    (*-ConfigBody)&15
00242                      rept    16-((*-ConfigBody)&15)
00243                      dc.b    0
```

```
00244                         endr
00245                         endc
00246                         else     others just need LONG-WORD
00247                         ifne     (*-ConfigBody)&3
00248                         rept     4-((*-ConfigBody)&3)
00249                         dc.b     0
00250                         endr
00251                         endc
00252                         endc     CPUTyp
00253                         endm
00254
00255
00256 * Configuration module body
00257 *
00258  00000000 ConfigBody equ      *
00259 0000 0000               dc.l     MaxMem          (unused)
00260 0004 002c               dc.w     PollSz          IRQ polling table
00261 0006 0045               dc.w     DevCnt          device table size
00262 0008 0040               dc.w     Procs           initial number of processes
00263 000a 0040               dc.w     Paths           initial number off paths
00264 000c 00b1               dc.w     SysParam        parameter string for first
executable module
00265 000e 00ab               dc.w     SysStart        first executable module name
offset
00266 0010 00b2              dc.w     SysDev         system default device name offset
00267 0012 00b6              dc.w     ConsolNm       standard I/O pathlist name offset
00268 0014 00c2               dc.w     Extens          Customization module name offset
00269 0016 00bc               dc.w     ClockNm         clock module name offset
00270 0018 0002               dc.w     Slice           number of ticks per time slice
00271 001a 0000               dc.w     IP_ID           interprocessor identification
00272 001c 0000               dc.l     Site            installation site code
00273 0020 009e               dc.w     MainFram        installation name offset
00274 0022 0001               dc.l     CPUTyp          specific 68000 family processor
in use
00275 0026 0103               dc.b     Level,Vers,Revis,Edit OS-9 Level
00276 002a 0090               dc.w     OS9Rev          OS-9 revision string offset
00277 002c 0080               dc.w     SysPri          initial system priority
00278 002e 0000              dc.w     MinPty         initial system minimum executable
priority
00279 0030 0000               dc.w     MaxAge          maximum system natural age limit
00280 0032 0064               dc.w     MDirSz          module directory count
00281 0034 0000               dc.w     0               reserved
00282 0036 0020              dc.w     Events         initial event table size (number
of entries)
00283 0038   10               dc.b     Compat          version change smooth byte
00284 0039   0f               dc.b     Compat2         cache control configuration
00285 003a 00f2               dc.w     MemList         memory definitions
00286 003c 0100               dc.w     StackSz/4       IRQ stack size (in longwords)
00287 003e 0000               dc.w     ColdRetrys      coldstart's "chd" retry count
00288 0040 0000               dc.w     0,0             reserved (MWKK)
00289 0044 0170               dc.w     CacheList       SSM(MMU) cache mode over-ride
list
00290 0046 01a4               dc.w     IOMan           IOMan module name
00291 0048 01aa               dc.w     PreIO           "PreIO" module list (called
before IOMan/Extens modules)
```

```
00292 004a 0000            dc.w     Config         system configuration control
00293 004c 0010           dc.w     NumSigs        maximum number of queued signals
00294 004e 05dc           dc.w     PrcDescStack   stack size in process descriptor
00295 0050 0000            dc.w     0,0,0,0,0,0,0,0 reserved
00296 0060 0000            dc.w     0,0,0,0,0,0,0,0 reserved
00297 0070 0000            dc.w     0,0,0,0,0,0,0,0 reserved
00298 0080 0000            dc.w     0,0,0,0,0,0,0,0 reserved
00299
00300
00301 * Configuration name strings
00302 *
00303 0090 4f53 OS9Rev     dc.b     "OS-9/68K V",Vers+'0',".",Revis+'0',0
00304
00305 * The remaining names are defined in the "systype.d" macro
00306 *
00307                      CONFIG
00308
00309 * default Extension module name lists (these are usually defined in
00310 * the system's systype.d file).
00311 * The calling sequence for Extension modules is:
00312 *   1. PreIO module(s)
00313 *   2. IOMan module(s)
00314 *   3. Extens (aka P2) module(s)
00315 *
00319
00321 01a4 494f IOMan      dc.b     "IOMan",0      default IOMan module name
00323
00325 01aa 4f53 PreIO      dc.b     "OS9PreIO",0   default "PreIO" module list
00326 * (these modules are called BEFORE the IOMan and Extens lists)
00328
00332
00336
00337 * define default caching modes (CPUTyp and system specific)
00338 * NOTE:  the following rules should be applied in determining
00339 *        the "coherency" of a cache and setting up the Compat2
00340 *        cache function flags:
00341 *
00342 *         - if the cache does not exits, then it is always coherent.
00343 *         - the on-chip cache coherency is not changable, except
00344 *           for the 68040.  If a 68040 system is used with
00345 *           bus-snooping disabled, then that fact should be registered
00346 *           by the user defining the label NoSnoop040 in their local
00347 *           "systype.d" file.
00348 *         - the coherency of external caches is indicated by the
00349 *           SnoopExt definition.  If the external caches are
00350 *           coherent or non-existant, then the label SnoopExt
00351 *           should be defined in "systype.d".
00352 *         - the kernel disables data caching when calling a file
00353 *           manager, unless the "NoDataDis" label is defined.
00354 *           Disabling data caching is required for systems possessing
00355 *           drivers using dma and don't perform any explicit data
00356 *           cache flushing.  If your system does NOT use dma drivers,
00357 *           or the drivers care for the cache, then the NoDataDis
00358 *           label should be defined in "systype.d".
00359 *
```

```
00361
00364 * external caches are coherent or absent
00365  00000003 ExtCache   equ      ExtC_I!ExtC_D
00370
00375  0000000f Compat2    set      ExtCache!OnC_I!OnC_D 68040 on-chip caches are
snoopy
00394
00396
00397  000001b4            ends
```

The Sysgo Module

The following is an example of the Sysgo module:

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/11/04   Page 1  sysgo.a
Sysgo - OS-9/68000 Initial (startup) module
00001                       nam      Sysgo
00002                       ttl      OS-9/68000     Initial (startup) module
00003
00015  00000004 Edition     equ      4              current edition number
00016
00017  00000101 Typ_Lang    set      (Prgrm<<8)+Objct
00018  00000000 Attr_Rev    set      0              (non-re- entrant)
00019                       psect    sysgo,Typ_Lang,Attr_Rev,Edition,128,Entry
00020
00021                       use      defsfile
00022
00023                       vsect
00024  00000000             ds.b     255            stack space
00025  00000000             ends
00026
00027 0000=4e40 Intercpt    os9      F$RTE          return from intercept
00028
00029 0004 41fa Entry       lea      Intercpt(pc),a0
00030 0008=4e40             os9      F$Icpt
00031 000c 41fa             lea      CmdStr(pc),a0  default execution dir ptr
00032 0010 7004             moveq    #Exec_,d0      execution mode
00033 0012=4e40             os9      I$ChgDir       chg exec dir (ignore errs)
00034 0016 640c             bcc.s    Entry10        continue if no error
00035 0018 7001             moveq    #1,d0          std output path
00036 001a 721a             moveq    #ChdErrSz,d1   size
00037 001c 41fa             lea      ChdErrMs(pc),a0 "Help, I can't find CMDS"
00038 0020=4e40             os9      I$WritLn       output error message
00039
00040 * Process startup file
00041 0024 7000 Entry10     moveq    #0,d0          std input path
00042 0026=4e40             os9      I$Dup          clone it
00043 002a 3e00             move.w   d0,d7          save cloned path number
00044 002c 7000             moveq    #0,d0          std input path
00045 002e=4e40             os9      I$Close
00046 0032 303c             move.w   #Read_,d0
00047 0036 41fa             lea      Startup(pcr),a0 "startup" pathlist
00048 003a=4e40             os9      I$Open         open startup file
00049 003e 640e             bcc.s    Entry15        continue if no error
00050 0040 7001             moveq    #1,d0          std output path
00051 0042 7220             moveq    #StarErSz,d1   size of startup error msg
00052 0044 41fa             lea      StarErMs(pc),a0 "Can't find 'startup'"
00053 0048=4e40             os9      I$WritLn       output error message
00054 004c 6032             bra.s    Entry25
00055
00056 004e 7000 Entry15     moveq    #0,d0          any type module
00057 0050 7200             moveq    #0,d1          no add'l default mem size
00058 0052 7406             moveq    #StartPSz,d2   sz of startup shell params
```

```
00059 0054 7603            moveq    #3,d3          copy three std I/O paths
00060 0056 7800            moveq    #0,d4          same priority
00061 0058 41fa            lea      ShellStr(pcr),a0 shell name
00062 005c 43fa            lea      StartPrm(pcr),a1 initial parameters
00063 0060=4e40            os9      F$Fork         fork shell
00064 0064 6410            bcc.s    Entry20        continue if no error
00065 0066 7001            moveq    #1,d0          std output path
00066 0068 7219            moveq    #FrkErrSz,d1   size
00067 006a 41fa            lea      FrkErrMs(pc),a0 "oh no, can't fork Shell"
00068 006e=4e40            os9      I$WritLn       output error message
00069 0072=4e40            os9      F$SysDbg       crash system
00070
00071 0076=4e40 Entry20    os9      F$Wait         wait for death,ignore error
00072 007a 7000            moveq    #0,d0          std input path
00073 007c=4e40            os9      I$Close        close redirected "startup"
00074 0080 3007 Entry25    move.w   d7,d0
00075 0082=4e40            os9      I$Dup          restore original std input
00076 0086 3007            move.w   d7,d0
00077 0088=4e40            os9      I$Close        remove cloned path
00078
00079 008c 7000 Loop       moveq    #0,d0          any type module
00080 008e 7200            moveq    #0,d1          default memory size
00081 0090 7401            moveq    #1,d2          one parameter byte (CR)
00082 0092 7603            moveq    #3,d3          copy std I/O paths
00083 0094 7800            moveq    #0,d4          same priority
00084 0096 41fa            lea      ShellStr(pcr),a0 shell name
00085 009a 43fa            lea      CRChar(pcr),a1 null paramter string
00086 009e=4e40            os9      F$Fork         fork shell
00087 00a2 650a            bcs.s    ForkErr        abort if error
00088 00a4=4e40            os9      F$Wait         wait for it to die
00089 00a8 6504            bcs.s    ForkErr
00090 00aa 4a41            tst.w    d1             zero status?
00091 00ac 67de            beq.s    Loop           loop if so
00092 00ae=4e40 ForkErr    os9      F$PErr         print error message
00093 00b2 60d8            bra.s    Loop
00094
00095 00b4 7368 ShellStr   dc.b     "shell",0
00096 00ba=5379 FrkErrMs   dc.b     "Sysgo can't fork 'shell'",C$CR
00097  00000019 FrkErrSz   equ      *-FrkErrMs
00098
00099 00d3 434d CmdStr     dc.b     "CMDS",0
00100 00d8=5379 ChdErrMs   dc.b     "Sysgo can't chx to 'CMDS'",C$CR
00101  0000001a ChdErrSz   equ      *-ChdErrMs
00102
00103 00f2 7374 Startup    dc.b     "startup",0
00104 00fa=5379 StarErMs   dc.b     "Sysgo can't open 'startup' file",C$CR
00105  00000020 StarErSz   equ      *-StarErMs
00106
00107 011a 2d6e StartPrm   dc.b     "-npxt"
00108 011f=  00 CRChar     dc.b     C$CR
00109  00000006 StartPSz   equ      *-StartPrm
00110  00000120            ends
```

Signals: Example Program

The following program demonstrates a subroutine reading a \n terminated string from a terminal with a ten second timeout between the characters. This program is designed to illustrate signal usage; it does not contain any error checking.

The `_ss_ssig(path, value)` library call notifies that operating system to send the calling process a signal with signal code value when data is available on `path`. If data is already pending, a signal is sent immediately. Otherwise, control returns to the calling program and the signal is sent when data arrives.

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

#define TRUE 1
#define FALSE 0

#define GOT_CHAR 2001
short dataready;     /* flag to show signal was received */

/* sighand - signal handling routine for this process */
sighand(signal)
register int signal;
{
     switch(signal) {
           /* ^E or ^C? */
           case 2:
           case 3:
                _errmsg(0,"termination signal received\n");
                exit(signal);
          /* Signal we're looking for? */
          case GOT_CHAR:
                dataready = TRUE;
                break;
          /* Anything else? */
          default:
                _errmsg(0,"unknown signal received ==> %d\n",signal);
                exit(1);
     }
}

main()
{
     char buffer[256];            /* buffer for typed-in string */

     intercept(sighand);          /* set up signal handler */

     printf("Enter a string:\n"); /* prompt user */
```

```
     /* call timed_read, returns TRUE if no timeout, -1 if timeout */
     if (timed_read(buffer) == TRUE)
          printf("Entered string = %s\n",buffer);
     else
          printf("\nType faster next time!\n");
}

int timed_read(buffer)
register char *buffer;
{
     char c = '\0';            /* 1 character buffer for read */
     short timeout = FALSE;    /* flag to note timeout occurred on read */
     int pos = 0;              /* position holder in buffer */

     /* loop until <return> entered or timeout occurs */
     while ( (c != '\n')  &&  (timeout == FALSE) ) {
          sigmask(1);            /* mask signals for signal setup */
          _ss_ssig(0,GOT_CHAR);  /* set up to have signal sent */
          sleep(10);             /* sleep for 10 seconds or until signal */

/* NOTE: we had to mask signals before doing _ss_ssig() so we did not get the
signal between the time we _ss_ssig()'ed and went to sleep.  */

          /* Now we're awake, determine what happened */
          if (!dataready)
               timeout = TRUE;
          else {
               read(0,&c,1);        /* read the ready byte */
               buffer[pos] = c;     /* put it in the buffer */
               pos++;               /* move our position holder */
               dataready = FALSE;   /* mark data as read */
          }
     }
     /* loop has terminated, figure out why */
     if (timeout)
          return -1;            /* there was a timeout so return -1 */
     else {
          buffer[pos] = '\0';   /* null terminate the string */
          return TRUE;
     }
}

#asm
* C binding for sigmask(value)
sigmask: move.l d1,-(sp)        save d1 on the stack
 move.l d0,d1                   get the passed parameter in the right place
 clr.l d0                       make d0 = 0
 os9 F$SigMask                  make the system call to mask signals
 bcc.s ret                      if no error...
 move.l #-1,d0                  return -1 to user
 move.l d1,errno(a6)            fill errno with error number
ret move.l (sp)+,d1             restore d1 from the stack
 rts                            return to user
#endasm
```

Alarms: Example Program

Compile the following example program with this command:

```
$ cc deton.c
```

The complete source code for the example program is as follows:

```
/*-------------------------------------------------------------------------|
|          Psect Name:deton.c                                              |
|          Function: demonstrate alarm to time out user input              |
|-------------------------------------------------------------------------*/
@_sysedit: equ 1

#include <stdio.h>
#include <errno.h>

#define TIME(secs) ((secs << 8) | 0x80000000)
#define PASSWORD "Ripley"

/*-------------------------------------------------------------------------*/
sighand(sigcode)
{
          /* just ignore the signal */
}

/*-------------------------------------------------------------------------*/
main(argc,argv)
int     argc;
char    **argv;
{
    register int    secs = 0;
    register int    alarm_id;
    register char   *p;
    register char   name[80];

    intercept(sighand);
    while (--argc)
        if (*(p = *(++argv)) == '-') {
            if (*(++p) == '?')
                printuse();
            else exit(_errmsg(1, "error: unknown option - '%c'\n", *p));
        } else if (secs == 0)
                secs = atoi(p);
            else exit(_errmsg(1, "unknown arg - \"%s\"\n", p));

    secs = secs ? secs : 3;
    printf("You have %d seconds to terminate self-destruct...\n", secs);


    /* set alarm to time out user input */
    if ((alarm_id = alm_set(2, TIME(secs))) == -1)
        exit(_errmsg(errno, "can't set alarm - "));
```

```
    if (gets(name) != 0)
        alm_delete(alarm_id);     /* remove the alarm; it didn't expire */
    else printf("\n");

    if (_cmpnam(name, PASSWORD, 6) == 0)
        printf("Have a nice day, %s.\n", PASSWORD);
    else printf("ka BOOM\n");

    exit(0);
}

/*---------------------------------------------------------------------------*/
/* printuse() - print help text to standard error                            */
printuse()
{
    fprintf(stderr, "syntax: %s [seconds]\n", _prgname());
    fprintf(stderr, "function: demonstrate use of alarm to time out I/O\n");
    fprintf(stderr, "options: none\n");
    exit(0);
}
```

Events: Example Program

The following program uses a binary semaphore to illustrate the use of events. To execute this example:

Step 1. Type the code into a file called `sema1.c`.

Step 2. Copy `sema1.c` to `sema2.c`.

Step 3. Compile both programs.

Step 4. Run both programs with this command: `sema1 & sema2.`

The program creates an event with an initial value of `1` (free), a wait increment of `-1`, and a signal increment of `1`. Then, the program enters a loop that waits on the event, prints a message, sleeps, and signals the event. After ten times through the loop, the program unlinks itself from the event and deletes the event from the system.

```
#include <stdio.h>
#include <events.h>
#include <errno.h>

char *ev_name = "semaevent";     /* name of event to be used */
int ev_id;     /* id used to access event */

main()
{
     int count = 0;  /* loop counter */

     /* create or link to the event */
     if ((ev_id = _ev_link(ev_name)) == -1)
          if ((ev_id = _ev_creat(1,-1,1,ev_name)) == -1)
               exit(_errmsg(errno,"error getting access to event - "));

     while (count++ < 10) {
          /* wait on the event */
          if (_ev_wait(ev_id, 1, 1) == -1)
               exit(_errmsg(errno,"error waiting on the event - "));

          _errmsg(0,"entering \"critical section\"\n");

          /* simulate doing something useful */
          sleep(2);
```

```
          _errmsg(0,"exiting \"critical section\"\n");

          /* signal event (leaving critical section) */
         if (_ev_signal(ev_id, 0) == -1)
               exit(_errmsg(errno,"error signalling the event - "));

          /* simulate doing something other than critical section */
          sleep(1);
     }
     /* unlink from event */
     if (_ev_unlink(ev_id) == -1)
          exit(_errmsg(errno,"error unlinking from event - "));

    /* delete event from system if this was the last process to unlink from it */
     if (_ev_delete(ev_name) == -1 && errno != E_EVBUSY)
          exit(_errmsg(errno,"error deleting event from system - "));

     _errmsg(0,"terminating normally\n");
}
```

Semaphores: Example Program

The following example shows how to use semaphores.

```
#ifndef _SEMAPHORE_H
#include <semaphore.h>
#endif
#ifndef _MODULE_H
#include <module.h>
#endif
register Semaphore sema;
Semaphore locate_semaphore();
/* link/create the semaphore */
sema = locate_semaphore();
while (1)  {
    /* perform semaphore "P" operation (reserve the semaphore) */
    if ((err = _os_sema_p(sema)) != SUCCESS)
        exit(_errmsg(err, "could not perform P operation - "));
    /* Enter critical section */
    /* perform semaphore "V" operation (release semaphore) */
    if ((err = _os_sema_v(sema)) != SUCCESS)
          exit(_errmsg(err, "could not perform V operation - "));
}
/* terminate usage of the semaphore */
_os_sema_term(sema);
}
#define ATTR_REV 0x8001  /* semaphore data-module's attribute revision value */
/* locate_semaphore - link or create semaphore module (initialize it). */
Semaphore locate_semaphore()
{
    register Semaphore sema;
    register mh_com *semamod;
    static char *semaname = "semaphore";
    mh_com *modlink();
    mh_com *_mkdata_module();
    /* attempt to link to the semaphore */
    if ((semamod = modlink(semaname, MT_DATA)) == ((mh_com*)-1))  {
        /* semaphore module did not exist so create it */
        if ((semamod = _mkdata_module("semaphore", sizeof semaphore, ATTR_REV,
                       MP_OWNER_READ|MP_OWNER_WRITE)) == ((mh_com*)(-1)))
            exit(_errmsg(errno, "can't create the semaphore - "));
        /* get the address of the semaphore data structure */
        sema = (Semaphore)((char*)semamod + semamod->m_exec);
        /* initialize the semaphore prior to usage the first time */
        _os_sema_init(sema);
    } else   {
        /* the semaphore module already exists */
        /* get the address of the semaphore data structure */
        sema = (Semaphore)((char*)semamod + semamod->m_exec);
    }
    return sema;
}
```

The following example shows how to use semaphores from assembly language.

```
*  Semaphore
 lea.l Semaphore(a1),a0
 move.l a0,d0 semaphore address
 bsr _os_sema_init
 move.l a0,d0 semaphore address
 bsr _os_sema_p
 move.l a0,d0 semaphore address
 bsr _os_sema_v
 move.l a1,d0 semaphore address
 bsr _os_sema_term
```

Note

Refer to the ***Using Ultra C*** manual for information about the `os_sema_xxx` call's operation and syntax.

C Trap Handler

Use the following makefile to make the example C trap handler and test programs:

```
# makefile - Used to make the example C trap handler and test program.

CFLAGS = -sqgixt=/dd
RDIR   = RELS
TRAP   = ctrap
TEST   = traptst

# Dependencies for making the entire example.

ctrap.example: $(TRAP) $(TEST)
    touch ctrap.example

# Dependencies for making the ctrap trap handler.

$(TRAP): tstart.r $(TRAP).r
    chd $(RDIR);\
    l68 tstart.r $(TRAP).r -l=/dd/lib/cio.l -l=/dd/lib/clib.l -l=/dd/lib/sys.l\
                -o=$(TRAP) -g

# Dependencies for making the traptst test program.

$(TEST): $(TEST).r

$(TEST).r: $(TEST).c
    cc -gim=2k $(TEST).c -r=$(RDIR)
```

The complete source for the C trap handler startup routines (`tstart.a`) is as follows:

```
****************************************
*
* tstart.a - C trap handler startup routines.
*
            nam     tstart C trap handler interface
            use     /dd/defs/oskdefs.d

*SYSTRAP    equ     1       define if trap should execute in system state

MaxParams   equ     20      maximum number of "C" style parameters allowed

        ifdef   SYSTRAP
AttrRevs    set     (ReEnt+SupStat)<<8  (system state)
        else
AttrRevs    set     (ReEnt)<<8          (user state)
        endc
```

```
TypeLang    set     (TrapLib<<8)+Objct
            psect   traphand,TypeLang,AttrRevs,0,0,TrapEnt
            dc.l    TrapInit
            dc.l    TrapTerm

*******************************
* Subroutine TrapInit
*   Trap handler initialization entry point
*
* Passed: d0.w  = User Trap number (1-15)
*         d1.l  = (optional) additional static storage
*         d2-d7 = caller's registers at time of trap
*         (a0)  = trap handler module name pointer
*         (a1)  = trap handler execution entry point
*         (a2)  = trap module pointer
*         a3-a5 = caller's registers at time of trap
*         (a6)  = trap handler static storage pointer
*         (a7)  = trap init stack frame pointer
*
* Returns: d0.l = "C" trapinit return value
*          (a0) = updated trap handler name pointer
*          (a1) = trap handler execution entry point
*          (a2) = trap module pointer
*          cc   = carry set, d1.w = error code if error
*          Other values returned are dependent on the trap handler
*
* The user stack looks like this:
*           .-------------------------.
*        +8 |    caller's return PC   |
*           |------------+------------|
*        +4 |    0000    |    0000    |
*           |------------|------------|
*           |   caller's a6 register  |
*   (usp)->  -------------------------
*
* NOTE: In system state, (a7)=system stack pointer.  This has a reasonable
*       amount of stack space (~1K).  No assumptions about where it is
*       should be made.

TrapInit:   bra     TrapEnt     call "C" trap handler (with func. code zero)

*******************************
* Subroutine TrapEnt
*   User Trap entry point
*
* Passed: d0-d7 = caller's registers
*         a0-a5 = caller's registers
*         (a6)  = trap handler static storage pointer
*         (a7)  = trap entry stack frame pointer
*         usp   = undisturbed user stack (in system state)
*
* Returns: cc   = carry set, d1.w=error code if error
*          Other values returned are dependent on the trap handler
*
* The system 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_CParams   do.l    MaxParams
S_a0        do.l    1           caller's a0 reg
S_a1        do.l    1           caller's a1 reg
S_a6        do.l    1           caller's a6 reg
S_func      do.w    1           trap function code
S_vect      do.w    1           user trap exception offset
S_cleanup   equ     .
S_pc        do.l    1           return pc

TrapEnt:    movem.l a0-a1,-(a7)         save regs
            lea     -MaxParams*4(a7),a7 allocate parameter space
            lea     S_CParams(a7),a1    ptr to C parameter area

        ifdef   SYSTRAP
            move    usp,a0      caller's parameters are on user stack ptr
            adda.l  #12,a0      above two rts pc's
        else
            lea     S_pc+16(a7),a0  caller's remaining C parameters ptr
        endc

            moveq   #MaxParams-1,d1 number of (potential) parameters
Trap10      move.l  (a0)+,(a1)+     copy caller's params from user stack
            dbra    d1,Trap10
            moveq   #0,d0           sweep reg
            move.w  S_func(a7),d0   1st param = func
            move.l  S_a6(a7),d1     2nd param = caller's (a6)
            bsr     ctrap           execute C traphandler
Trap90      movea.l S_a6(a7),a6     restore caller's a6
            lea     S_cleanup(a7),a7    discard scratch
            rts                     return to user program

*******************************
* Subroutine TrapTerm
*   Terminate trap handler servicing.
*
* As of this release (OS-9 V2.3) 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; so if it gets here...
            OS9     F$Exit          crash program (Error 001:199)

            ends
```

The complete source for the example C trap handler library (`ctrap.c`) is as follows:

```
/****************************************
 *
 * ctrap.c - Example C trap handler library.
 *
 * ctrap(func, a6, p1, p2, ...)
 */

int ctrap(func, a6, p1, p2, p3, p4)
register int func;              /* trap function code */
char *a6;                       /* caller's static storage base */
unsigned int p1, p2, p3, p4;    /* caller's parameters */
{
    register int result;

    switch(func)
    {
        case  0 :   result = 0;         break;  /* tlink call */
        case '+':   result = p1 + p2;   break;
        case '-':   result = p1 - p2;   break;
        case '*':   result = p1 * p2;   break;
        case '/':   result = p1 / p2;   break;
        case '&':   result = p1 & p2;   break;
        case '|':   result = p1 | p2;   break;
        case '^':   result = p1 ^ p2;   break;
        case '>':   result = p1 >> p2;  break;
        case '<':   result = p1 << p2;  break;
        default :   result = -1;        break;
    }
    return (result);
}
```

The complete source for `traptst.c`, which calls the `ctrap` handler, is as follows:

```
/*******************************************
 *
 * traptst.c - Calls the "ctrap" trap handler.
 *
 */

main()
{
    int   i, n;
    int   x = 22;
    int   y = 5;
    int   trapnum = 6;
    char *operator = "+-*/&|^<>?";

    printf("tlink: %d\n", tlink(trapnum, "ctrap"));

    n = strlen (operator);
```

```
    for (i = 0; i < n; ++i)
        printf("tcall(%d %c %d) = %d\n", x, operator[i], y,
            tcall(trapnum, operator[i], x, y));
}


/* bindings for tlink, tcall */
/*****************************************************************/
/* tlink(trapnum, trapname) - link to trap handler               */
/* int trapnum;             user trap number (1-15)              */
/* char *trapname;          name of trap module (NULL to unlink) */

#asm
tlink:      link    a5,#0
            movem.l a0-a2,-(a7)     save regs
            movea.l d1,a0           copy ptr to trap handler name
            moveq   #0,d1           no memory override
            OS9     F$TLink         link to trap handler
            bcc.s   tlink99         exit if no error
            move.l  d1,errno(a6)    save error number for caller
            moveq   #-1,d0          return error status
tlink99     movem.l (a7)+,a0-a2     restore regs
            unlk    a5
            rts
#endasm

/*****************************************************************/
/* tcall(trapnum, func, param1, param2, ...) - call trap handler */
/* int trapnum;             user trap number (1-15)              */
/* short func;              trap function number                 */
/* other parameters may be ints or pointers                      */

#asm
TRAP        equ     $4e40       user trap(0) opcode
RTS         equ     $4e75       rts opcode

            vsect
trapinst    ds.w    2
rtsinst     ds.w    1
            ends

tcall:      link    a5,#0
            tst.l   d0              valid trap number?
            beq.s   paramerr        abort if not
            cmp.l   #15,d0          valid trap number?
            bhi.s   paramerr
            add.w   #TRAP,d0
            movem.w d0-d1,trapinst(a6)  build usr trap instruction
            move.w  #RTS,rtsinst(a6)    set rts instruction
            moveq.l #0,d0           flush instruction cache
            os9     F$CCtl          ignore error
            jsr     trapinst(a6)    execute trap call
            bcc.s   tcall99         exit if no error
            move.l  d1,errno(a6)    save error number
            bra.s   tcallerr        abort
```

```
paramerr    move.l  #E$Param,errno(a6)
tcallerr    moveq   #-1,d0
tcall99
            unlk    a5
            rts
#endasm
```

RBF Device Descriptor

```
Microware OS-9/68020 Resident Macro Assembler V2.9  93/10/15  12:25  Page     1
 ../../../SRC/IO/RBF/DESC/d0.a
D0 Device Descriptor - Device Descriptor for Floppy disk controller
00001                       nam      D0 Device Descriptor
00002                       use      defsfile
00001
00002                       use      <oskdefs.d>
00001                       opt      -l
00004                       use      systype.d
00001 * System Definitions for MVME147 System
00002 *
00003                       opt      -l
00005
00006
00003
00004                       use      "rbfdesc.a"
00001
00002                       ttl      Device Descriptor for Floppy disk controller
00003
00004 * Copyright 1984, 1985, 1986, 1987, 1988, 1989, 1990, 1992, 1993
00005 * by Microware Systems Corporation.
00006 * Reproduced Under License.
00007
00008 * This source code is the proprietary confidential property of
00009 * Microware Systems Corporation, and is provided to licensee
00010 * solely for documentation and educational purposes. Reproduction,
00011 * publication, or distribution in any form to any party other
00012 * than the licensee is prohibited.
00050  00000011 Edition    equ      17             current edition number
00051
00052 * PD_DNS values
00053  00000000 Single     equ      0              FM encoded media
00054  00000001 Double     equ      1              MFM encoded media/double-track
density
00055  00000002 Quad       equ      1<<1           Quad track density
00056  00000004 Octal      equ      1<<2           Octal track density
00057
00058 * PD_TYP values
00059 * Note: For pre-V2.4 Five/Eight defines the disk size, rotational
00060 *       speed and data transfer rate.  From V2.4 the physical size
00061 *       is defined in bits 4 - 1, and PD_Rate defines the rotational
00062 *       speed and data transfer rate.
00063
00064 * floppy disk definitions
00065  00000000 Five       equ      0<<0           drive is 5 1/4"
00066  00000001 Eight      equ      1<<0           drive is 8"
00067  00000000 SizeOld    equ     0<<1          size/speed defined by bit 0 value
(pre-V2.4)
00068  00000002 Size8      equ      1<<1           physical size is 8"
00069  00000004 Size5      equ      2<<1           physical size is 5 1/4"
00070  00000006 Size3      equ      3<<1           physical size is 3 1/2"
```

```
00071
00072 * hard disk definitions
00073  00000040 HRemov     equ      1<<6           hard disk is removable
00074  00000080 Hard       equ      1<<7           hard disk media
00075
00076 * PD_Rate values
00077 * Note: V2.4 drivers should derive the disk data transfer rate and
00078 *       rotational speed from this field if PD_TYP, bits 4 - 1 are
00079 *       non-zero.  If not, then PD_TYP, bit 0 infers these.
00080  00000000 rpm300     equ      0              rotational speed is 300 rpm
00081  00000001 rpm360     equ      1              rotational speed is 360 rpm
00082  00000002 rpm600     equ      2              rotational speed is 600 rpm
00083  00000000 xfr125K    equ      0<<4           transfer rate is 125K bits/sec
00084  00000010 xfr250K    equ      1<<4           transfer rate is 250K bits/sec
00085  00000020 xrf300K    equ      2<<4           transfer rate is 300K bits/sec
00086  00000030 xfr500K    equ      3<<4           transfer rate is 500K bits/sec
00087  00000040 xfr1M      equ      4<<4           transfer rate is 1M bits/sec
00088  00000050 xfr2M      equ      5<<4           transfer rate is 2M bits/sec
00089  00000060 xfr5M      equ      6<<4           transfer rate is 5M bits/sec
00090
00091 * PD_VFY values
00092  00000001 ON         equ      1              "no-verify" ON
00093  00000000 OFF        equ      0             "no-verify" OFF (i.e. verify is
ON!)
00094
00095 * macro parameter #6 definitions (drive type)
00096
00097  00000001 d877       equ      1              single density 8"
00098  00000004 dd877      equ      4              double density 8"
00099  00000002 d540       equ      2              single density 5 1/4" 40 trk
00100  00000005 dd540      equ      5              double density 5 1/4" 40 trk
00101  00000003 d580       equ      3              single density 5 1/4" 80 trk
00102  00000006 dd580      equ      6              double density 5 1/4" 80 trk
00103  00000007 ramdisk    equ      7              volatile ram disk
00104  00000008 nvramdisk  equ      8              non-volatile ram disk
00105  00000009 uv580      equ      9              universal 5 1/4" 80 track
00106  0000000a autosize   equ      10            autosize device (SS_DSize tells
media size)
00107  0000000b dd380      equ      11             double density 3 1/2", 80 trk
00108  0000000c uv380      equ      12             universal 3 1/2" 80 track
00109  0000000d hd580      equ      13            double density 5 1/4" 80 track
'8" image'
00110  0000000e ed380      equ      14            double density 3 1/2" 80 track,
4M byte unformatted
00111  0000000f hd577      equ      15             double density 5 1/4" 77 track
'8" image'
00112  00000010 uv577      equ      16             universal 5 1/4" '8" image'
00113  00000011 uv877      equ      17             universal 8"
00114  00000012 hd380      equ      18            double density 3 1/2" 80 track,
2M (UF)/ 1.4M (F)
00115
00116  00000003 Density    set      BitDns+(TrkDns<<1)
00117  00000026 DiskType   set      DiskKind+(DnsTrk0<<5)
00118
00119  00000f00 TypeLang   set      (Devic<<8)+0
```

```
00120  00008000 Attr_Rev   set      (ReEnt<<8)+0
00121
00122                      psect    RBFDesc,TypeLang,Attr_Rev,Edition,0,0
00123
00124 0000 fffe            dc.l     Port           port address
00125 0004   45            dc.b     Vector         auto-vector trap assignment
00126 0005   04            dc.b     IRQLevel       IRQ hardware interrupt level
00127 0006   05            dc.b     Priority       irq polling priority
00128 0007   b7            dc.b     Mode           device mode capabilities
00129 0008 0048            dc.w     FileMgr        file manager name offset
00130 000a 004c            dc.w     DevDrv         device driver name offset
00131 000c 0053            dc.w     DevCon         (reserved)
00132 000e 0000            dc.w     0,0,0,0        reserved
00133 0016 0030            dc.w     OptLen
00134
00135 * Default Parameters
00136           OptTbl
00137 0018=  00            dc.b     DT_RBF         device type
00138 0019   00            dc.b     DrvNum         drive number
00139 001a   03            dc.b     StepRate       step rate
00140 001b   26            dc.b     DiskType       type of disk 8"/5 1/4"/Hard/etc
00141 001c   03            dc.b     Density        Bit Density and track density
00142 001d   00            dc.b     0              reserved
00143 001e 004f            dc.w     Cylnders-TrkOffs number of logical cylinders
00144 0020   02            dc.b     Heads          Number of Sides (Floppy)
Heads(Hard Disk)
00145 0021   01            dc.b     NoVerify       OFF = disk verify ON = no verify
00146 0022 0010            dc.w     SectTrk        default sectors/track
00147 0024 0010            dc.w     SectTrk0       default sectors/track track 0
00148 0026 0008            dc.w     SegAlloc       segment allocation size
00149 0028   01            dc.b     Intrleav       sector interleave factor
00150 0029   00            dc.b     DMAMode        DMA mode (driver dependant)
00151 002a   01            dc.b     TrkOffs        track base offset (first
accessable track)
00152 002b   01            dc.b     SectOffs       sector base offset (starting
physical sector number)
00153 002c 0100            dc.w     SectSize       # of bytes/sector
00154 002e 0002            dc.w     Control        control byte
00155 0030   07            dc.b     Trys           number of retrys 0 = no
retrys/error correction
00156 0031   00            dc.b     ScsiLun        scsi logical unit number
00157 0032 0000            dc.w     WrtPrecomp     write precomp cylinder
00158 0034 0000            dc.w     RedWrtCrnt     reduce write current cylinder
00159 0036 0000            dc.w     ParkCyl        cylinder to park head for hard
disk
00160 0038 0000            dc.l     LSNOffset      logical sector offset
00161 003c 0050            dc.w     TotalCyls      total cylinders on drive
00162 003e   06            dc.b     CtrlrID        scsi controller id
00163 003f   10            dc.b     Rates          data-transfer rate & rotational
speed
00164 0040 0000            dc.l     ScsiOpts       scsi option flags
00165 0044 00ff            dc.l     MaxCount-1     maximum byte count passable to
driver
00166  00000030 OptLen     equ      *-OptTbl
00167
```

```
00168 0048 5242 FileMgr    dc.b     "RBF",0        Random block file manager
00169           RBFDesc    macro
00170
00171           Port       equ      \1 Port address
00172           Vector     equ      \2 autovector number
00173           IRQLevel   equ      \3 hardware interrupt level
00174           Priority   equ      \4 polling priority
00175           DevDrv     dc.b     "\5",0 driver module name
00176                      ifgt     \#-5 standard device setup requested?
00177
00178
00179                      ifeq     \6-d877  8", 77 track drive, single density
00180           DiskKind   set      Eight+Size8 (set Eight for compatibility)
00181           Cylnders   set      77
00182           BitDns     set      Single FM encoding
00183           Rates      set      xfr250K+rpm360
00184           TrkDns     set      Single 48 tpi
00185           SectTrk    set      16
00186           SectTrk0   set      16
00187           TotalCyls  set      Cylnders number of actual cylinders on disk
00188                      endc
00189
00190                      ifeq     \6-dd877  8", 77 track, double density
00191           DiskKind   set      Eight+Size8 (set Eight for compatibility)
00192           Cylnders   set      77
00193           BitDns     set      Double MFM encoding
00194           Rates      set      xfr500K+rpm360
00195           TrkDns     set      Single 48 tpi
00196           SectTrk    set      28
00197           SectTrk0   set      16
00198           TotalCyls  set      Cylnders number of actual cylinders on disk
00199                      endc
00200
00201                     ifeq     \6-d540 5 1/4", 40 track drive, single density
00202           DiskKind   set      Size5
00203           Cylnders   set      40
00204           BitDns     set      Single FM encoding
00205           Rates      set      xfr125K+rpm300
00206           TrkDns     set      Single 48 tpi
00207           SectTrk    set      10
00208           SectTrk0   set      10
00209           TotalCyls  set      Cylnders number of actual cylinders on disk
00210                      endc
00211
00212                    ifeq     \6-dd540 5 1/4", 40 track, double density drive
00213           DiskKind   set      Size5
00214           Cylnders   set      40
00215           BitDns     set      Double MFM encoding
00216           Rates      set      xfr250K+rpm300
00217           TrkDns     set      Single 48 tpi
00218           SectTrk    set      16
00219           SectTrk0   set      10
00220           TotalCyls  set      Cylnders number of actual cylinders on disk
00221                      endc
00222
```

```
00223                     ifeq     \6-d580 5 1/4", 80 track, single density drive
00224           DiskKind  set      Size5
00225           Cylnders  set      80
00226           BitDns    set      Single FM encoding
00227           Rates     set      xfr125K+rpm300
00228           TrkDns    set      Double 96tpi
00229           SectTrk   set      10
00230           SectTrk0  set      10
00231           TotalCyls set      Cylnders number of actual cylinders on disk
00232                     endc
00233
00234                    ifeq     \6-dd580 5 1/4", 80 track drive, double density
00235           DiskKind  set      Size5
00236           Cylnders  set      80
00237           BitDns    set      Double MFM encoding
00238           Rates     set      xfr250K+rpm300
00239           TrkDns    set      Double 96tpi
00240           SectTrk   set      16
00241           SectTrk0  set      10
00242           TotalCyls set      Cylnders number of actual cylinders on disk
00243                     endc
00244
00245                     ifeq     \6-ramdisk volatile ram disk
00246           DiskKind  set      0
00247           Cylnders  set      0
00248           BitDns    set      Single
00249           TrkDns    set      Single
00250           SectTrk0  set      0
00251           Heads     set      0
00252           StepRate  set      0
00253           Intrleav  set      0
00254           NoVerify  set      ON
00255           SegAlloc  set      4
00256           Trys      set      0
00257           DevCon    set      0 not used by ram-disk driver
00258           Control   set      MultEnabl format enabled, m/s capable
00259           MaxCount  set      $ffffffff
00260                     endc
00261
00262                     ifeq     \6-nvramdisk non-volatile ram disk
00263           DiskKind  set      0
00264           Cylnders  set      0
00265           BitDns    set      Single
00266           TrkDns    set      Single
00267           SectTrk0  set      0
00268           Heads     set      0
00269           StepRate  set      0
00270           Intrleav  set      0
00271           NoVerify  set      ON
00272           SegAlloc  set      4
00273           Trys      set      0
00274           DevCon    set      0 not used by ram disk driver
00275           Control   set      FmtDsabl+MultEnabl nvram disks are format
disabled, m/s capable
00276           MaxCount  set      $ffffffff
```

```
00277                       endc
00278
00279                       ifeq     \6-uv580 universal 5 1/4" 80 track
00280           DiskKind    set      Size5   five inch disk
00281           Cylnders    set      80      number of (physical) tracks
00282           BitDns      set      Double MFM recording
00283           Rates       set      xfr250K+rpm300
00284           DnsTrk0     set      Double MFM track 0
00285           TrkDns      set      Double 96tpi
00286           SectTrk     set      16      sectors/track (except trk 0, side 0)
00287           SectTrk0    set      16      sectors/track, track 0, side 0
00288           SectOffs    set      1       physical sector start = 1
00289           TrkOffs     set      1       track 0 not used
00290           TotalCyls   set      Cylnders number of actual cylinders on disk
00291                       endc
00292
00293                       ifeq     \6-autosize "autosize" device (SS_DSize tells
capacity)
00294           SectTrk     set      0       sectors/track (except trk 0, side 0)
00295           SectTrk0    set      0       sectors/track, track 0, side 0
00296           Cylnders    set      0       total cylinders
00297           Heads       set      0       total heads
00298                       endc
00299
00300                       ifeq     \6-dd380 3 1/2", 80 track drive
00301           DiskKind    set      Size3
00302           Cylnders    set      80
00303           BitDns      set      Double
00304           Rates       set      xfr250K+rpm300
00305           TrkDns      set      Double 135tpi
00306           SectTrk     set      16
00307           SectTrk0    set      10
00308           TotalCyls   set      Cylnders number of actual cylinders on disk
00309                       endc
00310
00311                       ifeq     \6-uv380 universal 3 1/2" 80 track
00312           DiskKind    set      Size3
00313           Cylnders    set      80      number of (physical) tracks
00314           BitDns      set      Double MFM recording
00315           Rates       set      xfr250K+rpm300
00316           DnsTrk0     set      Double MFM track 0
00317           TrkDns      set      Double 135tpi
00318           SectTrk     set      16      sectors/track (except trk 0, side 0)
00319           SectTrk0    set      16      sectors/track, track 0, side 0
00320           SectOffs    set      1       physical sector start = 1
00321           TrkOffs     set      1       track 0 not used
00322           TotalCyls   set      Cylnders number of actual cylinders on disk
00323                       endc
00324
00325                       ifeq     \6-hd580 5 1/4" 80 track ’8" image’
00326           DiskKind    set      Eight+Size5 (set Eight for compatibility)
00327           Cylnders    set      80      number of (physical) tracks
00328           BitDns      set      Double MFM recording
00329           Rates       set      xfr500K+rpm360
00330           TrkDns      set      Double 96tpi
```

```
00331           SectTrk    set      28     sectors/track (except trk0, side 0)
00332           SectTrk0   set      16     sectors/track, track 0, side 0
00333           TotalCyls  set      Cylnders number of actual cylinders on disk
00334                      endc
00335
00336                      ifeq     \6-ed380 3 1/2" 80 track EXTRA density (4M
unformatted)
00337           DiskKind   set      Size3
00338           Cylnders   set      80     number of (physical) cylinders
00339           BitDns     set      Double MFM recording
00340           Rates      set      xfr1M+rpm300
00341           DnsTrk0    set      Double MFM track 0
00342           TrkDns     set      Double 135tpi
00343           SectTrk    set      61     sectors/track (except trk 0, side 0)
00344           SectTrk0   set      61     sectors/track, track 0, side 0
00345           SectOffs   set      1      physical sector start = 1
00346           TotalCyls  set      Cylnders number of actual cylinders on disk
00347                      endc
00348
00349                      ifeq     \6-hd577 5 1/4" 77 track ’8" image’
00350           DiskKind   set      Eight+Size5 (set Eight for compatibility)
00351           Cylnders   set      77     number of (physical) tracks
00352           BitDns     set      Double MFM recording
00353           Rates      set      xfr500K+rpm360
00354           TrkDns     set      Double 96tpi
00355           SectTrk    set      28     sectors/track (except trk0, side 0)
00356           SectTrk0   set      16     sectors/track, track 0, side 0
00357           TotalCyls  set      Cylnders number of actual cylinders on disk
00358                      endc
00359
00360                     ifeq     \6-uv577 universal 5 1/4" 77 track ’8" image’
00361           DiskKind   set      Eight+Size5 (set Eight for compatibility)
00362           Cylnders   set      77     number of (physical) tracks
00363           BitDns     set      Double MFM recording
00364           Rates      set      xfr500K+rpm360
00365           DnsTrk0    set      Double MFM track 0
00366           TrkDns     set      Double 96tpi
00367           SectTrk    set      28     sectors/track (except trk0, side 0)
00368           SectTrk0   set      28     sectors/track, track 0, side 0
00369           SectOffs   set      1      physical sector start = 1
00370           TrkOffs    set      1      track 0 not used
00371           TotalCyls  set      Cylnders number of actual cylinders on disk
00372                      endc
00373
00374                      ifeq     \6-uv877 universal 8" 77 track
00375           DiskKind   set      Eight+Size8 (set Eight for compatibility)
00376           Cylnders   set      77     number of (physical) tracks
00377           BitDns     set      Double MFM recording
00378           Rates      set      xfr500K+rpm360
00379           DnsTrk0    set      Double MFM track 0
00380           TrkDns     set      Single 48 tpi
00381           SectTrk    set      28     sectors/track (except trk0, side 0)
00382           SectTrk0   set      28     sectors/track, track 0, side 0
00383           SectOffs   set      1      physical sector start = 1
00384           TrkOffs    set      1      track 0 not used
```

```
00385           TotalCyls  set      Cylnders number of actual cylinders on disk
00386                      endc
00387
00388                     ifeq     \6-hd380 3 1/2" 80 track (2M unformatted, 1.4M
formatted)
00389           DiskKind   set      Eight+Size3 (set Eight for compatibility)
00390           Cylnders   set      80     number of (physical) tracks
00391           BitDns     set      Double MFM recording
00392           Rates      set      xfr500K+rpm300
00393           TrkDns     set      Double 96tpi
00394           SectSize   set      512    physical sector size
00395           SectTrk    set      18     sectors/track (except trk0, side 0)
00396           SectTrk0   set      18     sectors/track, track 0, side 0
00397           TotalCyls  set      Cylnders number of actual cylinders on disk
00398                      endc
00399
00400                      endc
00401                      endm
00402
00403 ***************
00404 * Descriptor Defaults
00405  000000b7 Mode       set      Dir_+ISize_+Append_+Exec_+Updat_
00406  00000000 BitDns     set      Single
00407  00000002 Heads      set      2
00408  00000002 StepRate   set      2
00409  00000003 Intrleav   set      3
00410  00000000 NoVerify   set      OFF
00411  00000000 DnsTrk0    set      Single
00412  00000000 DMAMode    set      0              non dma device
00413  00000008 SegAlloc   set      8              minimum segment allocation size
00414  00000000 TrkOffs    set      0
00415  00000000 SectOffs   set      0
00416  00000100 SectSize   set      256            default sector size 256 bytes.
00417  00000000 WrtPrecomp set      0              no write precomp
00418  00000000 RedWrtCrnt set      0              no reduced write current
00419  00000000 ParkCyl    set      0             where to park the head for hard
disk
00420  00000000 ScsiLun    set      0              scsi logical unit number
00421  00000000 CtrlrID    set      0              controller id
00422  00000000 LSNOffset  set      0              logical sector offset for scsi
hard disks
00423  00000000 TotalCyls  set      0              number of actual cylinders on
disk
00424
00425 * scsi options flag definitions
00426
00427  00000001 scsi_atn   set      1<<0           assert ATN supported
00428  00000002 scsi_target set      1<<1           target mode supported
00429  00000004 scsi_synchr set      1<<2          synchronous transfers supported
00430  00000008 scsi_parity set      1<<3           enable SCSI parity
00431
00432  00000000 ScsiOpts   set      0              scsi options flags (default)
00433
00434 * device control word definitions
00435
```

```
00436  00000000 FmtEnabl   set      0<<0           enable formatting
00437  00000001 FmtDsabl   set      1<<0           disable formatting
00438  00000000 MultDsabl  set      0<<1           disable multi-sectors
00439  00000002 MultEnabl  set      1<<1           enable multi-sectors
00440  00000000 StabDsabl  set      0<<2           device doesn't have stable id
00441  00000004 StabEnabl  set      1<<2           device has stable id
00442  00000000 AutoDsabl  set      0<<3           device size from device
descriptor
00443  00000008 AutoEnabl  set      1<<3           device tells size via SS_DSize
00444  00000000 FTrkDsabl  set      0<<4           device can't format a single
track
00445  00000010 FTrkEnabl  set      1<<4          device can format a single track
00446  00000000 WritEnab   set      0<<5           device is writable by RBF
00447  00000020 WritDsabl  set      1<<5          device is write-protected by RBF
00448  00000000 Control    set      0            descriptor control word (default)
00449
00450  00000007 Trys       set      7              number of Trys
00451  00010000 MaxCount   set      65536          default maximum transfer count
of driver (16-bit)
00452  00000000 Rates      set      0              default transfer-rate &
rotational speed
00453
00454 * end of file
00455
00005
00006  00000000 DrvNum     set      0
00007                      DiskD0
00008  0000005c            ends
```

SCF Device Descriptor

```
Microware OS-9/68020 Resident Macro Assembler V2.9  93/08/12  16:38  Page     1
 ../../../SRC/IO/SCF/DESC/term.a
Term - 68000 Term device descriptor module
00001                       nam      Term
00002                       ttl      68000 Term device descriptor module
00003                       use      defsfile
00001
00002                       use      <oskdefs.d>
00001                       opt      -l
00003                       use      systype.d
00001                       opt      -l
00004
00004
00005 * The default characteristics in "scfdesc.a" can be overridden
00006 * by equating the desired values here.  For example:
00007
00008                       use      "scfdesc.a"
00001                       ttl      Device Descriptor for SCF serial device
00002
00003 * Copyright 1983, 1984, 1985, 1988, 1991, 1992 by
00004 * 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
00048 0016 001c            dc.w     OptSiz         option byte count
00049
00050 * Default Parameters
00051           Options
00052 *                                        default
00053 *                                 name           function                 value
00054 *                                --------       --------------------     -------
00055 0018=  00            dc.b     DT_SCF         device type                SCF
00056 0019=  00            dc.b     upclock        upcase lock                OFF
00057 001a=  00            dc.b     bsb            backspace=BS,SP,BS         ON
00058 001b=  00            dc.b     linedel        line del/bsp line          OFF
00059 001c=  00            dc.b     autoecho       full duplex                ON
00060 001d=  00            dc.b     autolf         auto line feed             ON
00061 001e=  00            dc.b     eolnulls       null count                 0
00062 001f=  00            dc.b     pagpause       end of page pause          OFF
00063 0020=  00            dc.b     pagsize        lines per page             24
00064 0021=  00            dc.b     C$Bsp          backspace char             ^H
00065 0022=  00            dc.b     C$Del          delete line char           ^X
00066 0023=  00           dc.b     C$CR           end of record char       <return>
00067 0024=  00            dc.b     C$EOF          end of file char           ESC
00068 0025=  00            dc.b     C$Rprt         reprint line char          ^D
00069 0026=  00            dc.b     C$Rpet         dup last line char         ^A
```

```
00070 0027=  00           dc.b     C$Paus         pause char               ^W
00071 0028=  00           dc.b     C$Intr         Keyboard Interrupt char  ^C
00072 0029=  00           dc.b     C$Quit         Keyboard Quit char       ^E
00073 002a=  00           dc.b     C$Bsp          backspace echo char      ^H
00074 002b=  00           dc.b     C$Bell         line overflow char       ^G
00075 002c   00           dc.b     Parity         stop bits and parity     none
00076 002d   0e           dc.b     BaudRate       bits/char and baud rate  none
00077 002e=0000           dc.w     EchoNam        offset of echo device    none
00078 0030=  00           dc.b     C$XOn          Transmit Enable char     ^Q
00079 0031=  00           dc.b     C$XOff         Transmit Disable char    ^S
00080 0032=  00           dc.b     C$Tab          tab character            ^I
00088  0000001c OptSiz     equ      *-Options
00089
00090 0034 5363 FileMgr    dc.b     "Scf",0        file manager
00091
00092 * Macro to generate main features of device descriptor
00093          SCFDesc    macro
00094                     ifne     \#-7 must have exactly seven arguments
00095                     fail     SCFDesc: must specify all 7 arguments
00096                     endc
00097
00098          Port       equ      \1 Port address
00099          Vector     equ      \2 autovector number
00100          IRQLevel   equ      \3 hardware interrupt level
00101          Priority   equ      \4 polling priority
00102          Parity     equ      \5 parity, stop bits
00103          BaudRate   equ      \6 baud rate
00104          DevDrv     dc.b     "\7",0 driver module name
00105          EchoNam    equ      bname echo device descriptor (self)
00106
00107                     ifdef    KANJI
00108                     ifndef   Kinit
00109          Kinit      set      0default values
00110                     endc
00111                     ifndef   Kreset
00112          Kreset     set      0
00113                     endc
00114                     ifndef   Kin
00115          Kin        set      0
00116                     endc
00117                     ifndef   Kout
00118          Kout       set      0
00119                     endc
00120                     endc     KANJI
00121
00122                     endm
00123
00124  00000023 Mode       set      ISize_+Updat_  default device mode capabilities
00009
00010                     TERM
00011  00000040           ends
00012
```

SBF Device Descriptor

```
Microware OS-9/68020 Resident Macro Assembler V2.9  93/08/12  16:40  Page     1
 ../../../SRC/IO/SBF/DESC/mt0.a
MT0 Device Descriptor - LRCChip.d - Local Resource Controller definitions
00001                       nam       MT0 Device Descriptor
00002                       use       defsfile
00001
00002                       use       <oskdefs.d>
00001                       opt       -l
00003                       use       systype.d
00001                       opt       -l
00004
00003
00004                       use       "sbfdesc.a"
00001                       ttl       Device Descriptor for Tape controller
00002
00003 * Copyright 1986, 1988, 1989, 1993 by Microware Systems Corporation.
00004 * Reproduced Under License.
00005
00006 * This source code is the proprietary confidential property of
00007 * Microware Systems Corporation, and is provided to licensee
00008 * solely for documentation and educational purposes. Reproduction,
00009 * publication, or distribution in any form to any party other
00010 * than the licensee is prohibited.
00011
00035  00000f00 TypeLang   set       (Devic<<8)+0
00036  00008000 Attr_Rev   set       (ReEnt<<8)+0
00037
00038                       psect     SBFDesc,TypeLang,Attr_Rev,Edition,0,0
00039
00040 0000 ffff            dc.l      Port            port address
00041 0004   bd            dc.b      Vector          vector trap assignment
00042 0005   02            dc.b      IRQLevel        IRQ hardware interrupt level
00043 0006   05            dc.b      Priority        irq polling priority
00044 0007   67            dc.b      Mode            device mode capabilities
00045 0008 002c            dc.w      FileMgr         file manager name offset
00046 000a 0038            dc.w      DevDrv          device driver name offset
00047 000c 0030            dc.w      DevCon          device constants offset
00048 000e 0000            dc.w      0,0,0,0         reserved
00049 0016 0014            dc.w      OptLen
00050
00051 * Default Parameters
00052  00000018 OptTbl     equ       *
00053 0018=  00            dc.b      DT_SBF          device type
00054 0019   00            dc.b      DrvNum          drive number
00055 001a   00            dc.b      0               reserved
00056 001b   64            dc.b      NumBlks         maximum number of block buffers
00057 001c 0000            dc.l      BlkSize         block size
00058 0020 03e8            dc.w      DrvPrior        driver process priority
00059 0022   00            dc.b      SBFFlags        file manager flags
00060 0023   00            dc.b      DrivFlag        driver flags
00061 0024 0000            dc.w      DMAMode         DMA type/usage
```

```
00062 0026   04            dc.b     ScsiID         controller ID on SCSI bus
00063 0027   00            dc.b     ScsiLUN        tape drive LUN on controller
00064 0028 0000            dc.l     ScsiOpts       scsi option flags
00065  00000014 OptLen     equ      *-OptTbl
00066
00067 002c 5342 FileMgr    dc.b     "SBF",0        Sequential Block File manager
00068
00069 * SBFDesc Macro definitions
00070 *
00071           SBFDesc    macro
00072
00073           Port       equ      \1 Port address
00074           Vector     equ      \2 autovector number
00075           IRQLevel   equ      \3 hardware interrupt level
00076           Priority   equ      \4 polling priority
00077           DevDrv     dc.b     "\5",0 driver module name
00078
00079                      ifgt     \#-5 standard device setup requested?
00080 * reserved for future "std" options
00081
00082                      endc
00083                      endm
00084
00085 ****************
00086 * Descriptor Defaults
00087  00000067 Mode       set      Share_+ISize_+Exec_+Updat_
00088  00000000 Speed      set      0              driver defined
00089  00000002 NumBlks    set      2             (0=unbuffered mode, else number
of buffers)
00090  00002000 BlkSize    set      0x2000
00091  000003e8 DrvPrior   set      1000           "sbf process" priority
00092  00000000 SBFFlags   set      0
00093  00000000 DrivFlag   set      0
00094  00000000 DMAMode    set      0              driver defined
00095  00000000 ScsiID     set      0
00096  00000000 ScsiLUN    set      0
00097
00098 * scsi options flag definitions
00099
00100  00000001 scsi_atn    set      1<<0           assert ATN supported
00101  00000002 scsi_target set      1<<1           target mode supported
00102  00000004 scsi_synchr set      1<<2          synchronous transfers supported
00103  00000008 scsi_parity set      1<<3           enable SCSI parity
00104  00000000 ScsiOpts    set      0              scsi options flags
00105
00106 * end of file
00107
00108
00005
00006  00000000 DrvNum     set      0
00007                      TapeMT0
00008  00000040            ends
```

Pipe Device Descriptor

```
Microware OS-9/68020 Resident Macro Assembler V2.9  94/05/05  13:20  Page     1
 pipe.a
Pipe device descriptor module -
00001                       nam      Pipe device descriptor module
00016
00017                       use      defsfile
00001
00002                       use      <oskdefs.d>
00001                       opt      -l
00003
00018
00019  00000f00 TypeLang    set      (Devic<<8)+0
00020  00008000 Attr_Rev    set      (ReEnt<<8)+0
00021                       psect    Pipe,TypeLang,Attr_Rev,Edition,0,0
00022
00023 0000 0000             dc.l     0              no port address
00024 0004   00             dc.b     0              no trap assignment
00025 0005   00             dc.b     0              no IRQ hardware interrupt level
00026 0006   00             dc.b     0              0 no polling priority
00027 0007   a7             dc.b     Dir_+ISize_+Exec_+Updat_ device mode
capabilities
00028 0008 001e             dc.w     PipeMgr        file manager name offset
00029 000a 0026             dc.w     PipeDrv        device driver name offset
00030 000c 0000             dc.w     0              DevCon
00031 000e 0000             dc.w     0,0,0,0        reserved
00032 0016 0006             dc.w     OptLen         option byte count

00033
00034           OptTbl
00035 0018=  00             dc.b     DT_Pipe        Device Type: Pipe
00036 0019   00             dc.b     0              reserved
00037 001a 0000             dc.l     0              pipe buffer size (zero=default
tiny buffer)
00038  00000006 OptLen      equ      *-OptTbl
00039
00040 001e 5069 PipeMgr     dc.b     "PipeMan",0    file manager
00041 0026 4e75 PipeDrv     dc.b     "Null",0       device driver
00042  0000002c             ends
00043
```

Appendix B: Path Descriptors and Device Descriptors

This appendix includes the device descriptor initialization table definitions and path descriptor option tables for RBF, SCF, SBF, and PIPEMAN type devices. Refer to Appendix A: Example Code for RBF, SCF, SBF, and pipe example device descriptors.

This appendix contains the following topics:

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 3: OS-9 Input/Output System for full descriptions). **Table B-2** shows a graphic representation of the table. The size of the table is defined in the `M$Opt` field.

Table B-1 RBF Device Descriptor Modules

Name	Description
`PD_DTP`	**Device Type** This field is set to one for RBF devices. (0=SCF, 1=RBF, 2=PIPE, 3=SBF, 4=NET)
`PD_DRV`	**Drive Number** Use this field to associate a one-byte logical integer with each drive a driver/ controller handles. Number each controller's drives `0` to `n-1` (n is the maximum number of drives the controller can handle and is set into `V_NDRV` by the driver's `INIT` routine). This number defines which drive table the driver and RBF access for this device. RBF uses this number to set up the drive table pointer (`PD_DTB`). Before initializing `PD_DTB`, RBF verifies `PD_DRV` is valid for the driver by checking for a value less than `V_NDRV` in the driver's static storage. If not valid, RBF aborts the path open and returns an error. On simple hardware, this logical drive number is often the same as the physical drive number.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_STP	**Step Rate** PD_STP contains a code that sets the drive's head-stepping rate. To reduce access time, set the step rate to the fastest value of which the drive is capable. For floppy disks, the following codes are commonly used: **Step Code** — **5" Disks** — **8" Disks** 0 — 30ms — 15ms 1 — 20ms — 10ms 2 — 12ms — 6ms 3 — 6ms — 3ms For hard disks, the value in this field is usually driver dependent.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_TYP	**Disk Type** Defines the physical type of the disk, and indicates the revision level of the descriptor. If bit 7 = 0, floppy disk parameters are described in bits 0-6: bit 0: 0 = 5 1/4" floppy disk (pre-Version 2.4 of OS-9 for 68K) 1 = 8" floppy disk (pre-Version 2.4 of OS-9 for 68K) bits 1-3: 0 = (pre-OS-9 for 68K Version 2.4 descriptor) Bit 0 describes type/rates. 1 = 8" physical size 2 = 5 1/4" physical size 3 = 3 1/2" physical size 4-7: Reserved bit 4: Reserved bit 5: 0 = Track 0, side 0, single density 1 = Track 0, side 0, double density bit 6: Reserved If bit 7 = 1, hard disk parameters are described in bits 0-6: bits 0-5: Reserved bit 6: 0 = Fixed hard disk 1 = Removable hard disk

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_DNS	**Disk Density** The hardware density capabilities of a floppy disk drive: bit 0: 0 = Single bit density (FM) 1 = Double bit density (MFM) bit 1: 1 = Double track density (96 TPI/135 TPI) bit 2: 1 = Quad track density (192 TPI) bit 3: 1 = Octal track density (384 TPI) This parameter is format specific.
PD_CYL	**Logical Number of Cylinders (Tracks)** The logical number of cylinders per disk. format uses this value, PD_SID, and PD_SCT to determine the size of the drive. PD_CYL is often the same as the physical cylinder count (PD_TotCyls), but can be smaller if using partitioned drives (PD_LSNOffs) or track offsetting (PD_TOffs). If the drive is an autosize drive (PD_Cntl), format ignores this field. This parameter is format specific.
PD_SID	**Heads or Sides** The number of heads for a hard disk (Heads) or the number of surfaces for a floppy disk (Sides). If the drive is an autosize drive (PD_Cntl), format ignores this field. This parameter is format specific.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_VFY	**Verify Flag** Indicates whether or not to verify write operations. 0 = verify disk write 1 = no verification **NOTE:** Write verify operations are generally performed on floppy disks. They are not generally performed on hard disks because of the lower soft error rate of hard disks.
PD_SCT	**Default Number of Sectors/Track** If the drive is an autosize drive (PD_Cntl), format ignores this field. This parameter is format specific.
PD_T0S	**Default Sectors/Track (Track 0)** The number of sectors per track for track 0. This may be different than PD_SCT depending on specific disk format. If the drive is an autosize drive (PD_Cntl), format ignores this field. This parameter is format specific.
PD_SAS	**Segment Allocation Size** The default minimum number of sectors to allocate when a file is expanded. Typically, this is set to the number of sectors on the media track (for example, 8 for floppy disks, 32 for hard disks), but you can adjust it to suit your system's requirements.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_ILV	**Sector Interleave Factor** The sequential arrangement of sectors on a disk (for example, 1, 2, 3... or 1, 3, 5...). For example, if the interleave factor is 2, the sectors are arranged by 2's (1, 3, 5...) starting at the base sector (see PD_SOffs). **NOTE:** Optimized interleaving can drastically improve I/O throughput. **NOTE:** PD_ILV is typically only used when the media is formatted, as format uses this field to determine the default interleave. However, when the media format occurs (I$SetStt, SS_WTrk call), the desired interleave is passed in the parameters of the call. This parameter is format specific.
PD_TFM	**DMA (Direct Memory Access) Transfer Mode** The mode of transfer for DMA access, if the driver can handle different DMA modes. Use of this field is driver dependent.
PD_TOffs	**Track Base Offset** The offset to the first accessible physical track number. Track 0 is not always used as the base track because it is often a different density. This parameter is format specific.
PD_SOffs	**Sector Base Offset** The offset to the first accessible physical sector number on a track. Sector 0 is not always the base sector. This parameter is format specific.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_SSize	**Physical Sector Size in Bytes** The default sector size is 256. Depending on whether the driver supports non-256 byte logical sector sizes (a variable sector size driver), PD_SSize is used as follows: • **Variable Sector Size Driver** If the driver supports variable logical sector sizes, RBF inspects this value during a path open (specifically, after the driver returns **no error** on the SS_VarSect GetStat call) and uses this value as the ***logical*** sector size of the media. This value is then copied into PD_SctSiz of the path descriptor options section, so application programs can know the logical sector size of the media, if required. RBF supports logical sector sizes from 256 bytes to 32,768 bytes, in integral binary multiples (256, 512, 1024, etc.). During the SS_VarSect call, the driver can validate or update this field (or the media itself) according to the driver's conventions. These typically are: • If the driver can dynamically determine the media's sector size, and PD_SSize is passed in as 0, the driver updates this field according to the current media setting.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_SSize (continued)	• If the driver can dynamically set the media's sector size, and PD_SSize is passed in as a non-zero value, the driver sets the media to the value in PD_SSize (this is typical when reformatting the media). • If the driver cannot dynamically determine or set the media sector size, it usually validates PD_SSize against the supported sector sizes, and returns an error (E$SectSiz) if PD_SSize contains an invalid value. • **Non-Variable Sector Size Driver** If the driver does not support variable logical sector sizes (logical sector size is fixed at 256 bytes), RBF ignores PD_SSize. In this case, you can use PD_SSize to support deblocking drivers that support various physical sector sizes. **NOTE:** A non-variable sector sized driver is defined as a driver returning the E$UnkSvc error for GetStat (SS_VarSect).

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_Cntl	**Device Control Word** Indicates options reflecting the device's capabilities. You may set these options, as follows: bit 0: 0 = Format enable 1 = Format inhibit bit 1: 0 = Single-Sector I/O 1 = Multi-Sector I/O capable bit 2: 0 = Device has non-stable ID 1 = Device has stable ID bit 3: 0 = Device size determined from descriptor values 1 = Device size obtained by SS_DSize GetStat call bit 4: 0 = Device cannot format a single track 1 = Device can format a single track bit 5: 0 = Media is writable by RBF. 1 = Media is write-protected by RBF. bits 6-15: Reserved

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_Trys	**Number of Tries** Indicates whether a driver should try to access the disk again before returning an error. Depending on the driver in use, this field may be implemented as a flag or a retry counter: **Value** / **Flag** / **Counter** 0 / retry ON / Default # of retries 1 / retry OFF / No retries other / retry ON / Specified # of retries Drivers working with controllers having error correcting functions (for example, E.C.C. on hard disks) should treat PD_Trys as a flag so they can set the controller's error correction/retry functions accordingly. When formatting media, especially hard disks, the format-enabled descriptor should set this field to one (retry OFF) to ensure marginal media sections are marked out of the media free space.
PD_LUN	**Logical Unit Number of SCSI Drive** Used in the SCSI command block to identify the logical unit on the SCSI controller. To eliminate allocation of unused drive tables in the driver static storage, this number may be different from PD_DRV. PD_DRV indicates the logical number of the drive to the driver (the drive table to use). PD_LUN is the physical drive number on the controller.
PD_WPC	**First Cylinder to Use Write Precompensation**

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_RWR	**First Cylinder to Use Reduced Write Current**
PD_Park	**Cylinder Used to Park Head** The cylinder at which to park the hard disk's head when the drive is shut down. Parking is usually done on hard disks when they are shipped or moved and is implemented by the SS_SQD SetStat to the driver.
PD_LSNOffs	**Logical Sector Offset** The offset to use when accessing a partitioned drive. The driver adds this value to the logical block address passed by RBF prior to determining the physical block address on the media. Typically, using PD_LSNOffs is mutually exclusive to using PD_TOffs.
PD_TotCyls	**Total Cylinders on Device** The actual number of physical cylinders on a drive. It is used by the driver to correctly initialize the controller/drive. PD_TotCyls is typically used for physical initializing a partitioned drive or one with PD_TOffs set to a non-zero value. In this case, PD_CYL denotes the ***logical*** number of cylinders of the drive. If PD_TotCyls is 0, the driver should determine the physical cylinder count by using the sum of PD_CYL and PD_TOffs.
PD_CtrlrID	**SCSI Controller ID** The ID number of the SCSI controller attached to the drive. The driver uses this number to communicate with the controller.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_ScsiOpt	**SCSI Driver Options Flags** The SCSI device options and operation modes. It is the driver's responsibility to use or reject these values, as applicable. bit 0: 0 = ATN not asserted (no disconnect allowed) 1 = ATN asserted (disconnect allowed) bit 1: 0 = Device cannot operate as a target 1 = Device can operate as a target bit 2: 0 = Asynchronous data transfer 1 = Synchronous data transfer bit 3: 0 = Parity off 1 = Parity on All other bits are reserved.

Table B-1 RBF Device Descriptor Modules (continued)

Name	Description
PD_Rate	**Data Transfer/Rotational Rate** The data transfer rate and rotational speed of the floppy media. PD_Rate is normally used only when the physical size field (PD_TYP, bits 1-3) is non-zero. bits 0-3: Rotational speed 0 = 300 RPM 1 = 360 RPM 2 = 600 RPM All other values are reserved. bits 4-7: Data transfer rate 0 = 125K bits/sec 1 = 250K bits/sec 2 = 300K bits/sec 3 = 500K bits/sec 4 = 1M bits/sec 5 = 2M bits/sec 6 = 5M bits/sec All other values are reserved.
PD_MaxCnt	**Maximum Transfer Count** The maximum byte count the driver can transfer in one call. If this field is 0, RBF defaults to the value of $ffff (65,535).

Note

Offset in the following table 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`.

Table B-2 Initialization Table for RBF Device Descriptor Modules

Device Descriptor Offset	Path Descriptor Label	Description
$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	Number of Cylinders
$50	PD_SID	Number of Heads/Sides
$51	PD_VFY	Disk Write Verification
$52	PD_SCT	Default Sectors/Track

Table B-2 Initialization Table for RBF Device Descriptor Modules (continued)

Device Descriptor Offset	Path Descriptor Label	Description
$54	PD_T0S	Default Sectors/Track 0
$56	PD_SAS	Segment Allocation Size
$58	PD_ILV	Sector Interleave Factor
$59	PD_TFM	DMA Transfer Mode
$5A	PD_TOffs	Track Base Offset
$5B	PD_SOffs	Sector Base Offset
$5C	PD_SSize	Sector Size (in bytes)
$5E	PD_Cntl	Control Word
$60	PD_Trys	Number of Tries
$61	PD_LUN	SCSI Unit Number of Drive
$62	PD_WPC	Cylinder to Begin Write Precompensation
$64	PD_RWR	Cylinder to Begin Reduced Write Current
$66	PD_Park	Cylinder to Park Disk Head
$68	PD_LSNOffs	Logical Sector Offset
$6C	PD_TotCyls	Number of Cylinders On Device

Table B-2 Initialization Table for RBF Device Descriptor Modules (continued)

Device Descriptor Offset	Path Descriptor Label	Description
$6E	PD_CtrlrID	SCSI Controller ID
$6F	PD_Rate	Data transfer/Disk Rotation Rates
$70	PD_ScsiOpt	SCSI Driver Options Flags
$74	PD_MaxCnt	Maximum Transfer Count

RBF Definitions of the Path Descriptor

The first 26 fields of the path options section (PD_OPT) of the RBF path descriptor are copied directly from the device descriptor standard initialization table. All of the values in this table may be examined using I$GetStt by applications using the SS_Opt code. Some of the values may be changed using I$SetStt; some are protected by the file manager to prevent inappropriate changes. You can update the following fields using GetStat and SetStat system calls:

- PD_STP
- PD_TYP
- PD_DNS
- PD_CYL
- PD_SID
- PD_VFY
- PD_SCT
- PD_TOS
- PD_SAS

All other fields are read-only. The RBF path descriptor option table is shown on the following page.

Refer to the previous section on RBF device descriptors for descriptions of the first 26 fields. The last five fields contain information provided by RBF:

Table B-3 RBF Path Descriptor Option

Name	Description
PD_ATT	**File Attributes** (D S PE PW PR E W R) The file's attributes are defined as follows: bit 0: Set if owner read. bit 1: Set if owner write. bit 2: Set if owner execute. bit 3: Set if public read. bit 4: Set if public write. bit 5: Set if public execute. bit 6: Set if only one user at a time can open the file. bit 7: Set if directory file.
PD_FD	**File Descriptor** The LSN (Logical Sector Number) of the file's file descriptor is written here.
PD_DFD	**Directory File Descriptor** The LSN of the file's directory file descriptor is written here.
PD_DCP	**File's Directory Entry Pointer** The current position of the file's entry in its directory.
PD_DVT	**Device Table Pointer (Copy)** The address of the device table entry associated with the path.

Table B-3 RBF Path Descriptor Option (continued)

Name	Description
PD_SctSiz	**Logical Sector Size** The logical sector size of the device associated with the path. If this is 0, assume a size of 256 bytes.
PD_NAME	**File Name**

Note

In the following table, offset refers to the location of a path descriptor field relative to the starting address of the path descriptor. Path descriptor offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: `sys.l` or `usr.l`.

Table B-4 Option Table for RBF Path Descriptor

Offset	Name	Description
$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

Table B-4 Option Table for RBF Path Descriptor (continued)

Offset	Name	Description
$86	PD_CYL	Number of Cylinders
$88	PD_SID	Number of Heads/Sides
$89	PD_VFY	Disk Write Verification
$8A	PD_SCT	Default Sectors/Track
$8C	PD_TOS	Default Sectors/Track 0
$8E	PD_SAS	Segment Allocation Size
$90	PD_ILV	Sector Interleave Factor
$91	PD_TFM	DMA Transfer Mode
$92	PD_TOffs	Track Base Offset
$93	PD_SOffs	Sector Base Offset
$94	PD_SSize	Sector Size (in bytes)
$96	PD_Cntl	Control Word
$98	PD_Trys	Number of Tries
$99	PD_LUN	SCSI Unit Number of Drive
$9A	PD_WPC	Cylinder to Begin Write Precompensation
$9C	PD_RWR	Cylinder to Begin Reduced Write Current

Table B-4 Option Table for RBF Path Descriptor (continued)

Offset	Name	Description
$9E	PD_Park	Cylinder to Park Disk Head
$A0	PD_LSNOffs	Logical Sector Offset
$A4	PD_TotCyls	Number of Cylinders On Device
$A6	PD_CtrlrID	SCSI Controller ID
$A7	PD_Rate	Data Transfer/Rotational Rates
$A8	PD_ScsiOpt	SCSI Driver Option Flag
$AC	PD_MaxCnt	Maximum Transfer Count
$B0		Reserved
$B5	PD_ATT	File Attributes
$B6	PD_FD	File Descriptor
$BA	PD_DFD	Directory File Descriptor
$BE	PD_DCP	File's Directory Entry Pointer
$C2	PD_DVT	Device Table Pointer (copy)
$C6		Reserved
$C8	PD_SctSiz	Logical Sector Size
$CC		Reserved
$E0	PD_NAME	File Name

SCF Device Descriptor Modules

Device descriptor modules for SCF-type devices contain the device address and an initialization table which defines initial values for the I/O editing features, as listed below. The initialization table immediately follows the standard device descriptor module header fields (refer to Chapter 3: OS-9 Input/Output System for full descriptions). The size of the table is defined in the `M$Opt` field. The initialization table is graphically shown in **Table B-6** and the following table.

Note

You can change or disable most of these special editing functions by changing the corresponding control character in the path descriptor. You can do this with the `I$SetStt` service request or the `tmode` utility. A permanent solution may be to change the corresponding control character value in the device descriptor module. You can easily change the device descriptors with the `xmode` utility.

Table B-5 SCF Device Descriptor Modules

Name	Description
`PD_DTP`	**Device Type** Set to `0` for SCF devices. (`0`=SCF, `1`=RBF, `2`=PIPE, `3`=SBF, `4`=NET)
`PD_UPC`	**Letter Case** If `PD_UPC` is not equal to `0`, input or output characters in the range `a..z` are made `A..Z`.

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_BSO	**Destructive Backspace** If PD_BSO is 0 when a backspace character is input, SCF echoes PD_BSE (backspace echo character). If PD_BSO is non-zero, SCF echoes PD_BSE, space, PD_BSE.
PD_DLO	**Delete** If PD_DLO is 0, SCF deletes by backspace-erasing over the line. If PD_DLO is non-zero, SCF deletes by echoing a carriage return/line-feed.
PD_EKO	**Echo** If PD_EKO is non-zero, all input bytes are echoed, except undefined control characters, which are printed as periods. If PD_EKO is 0, input characters are not echoed.
PD_ALF	**Automatic Line Feed** If PD_ALF is non-zero, line-feeds automatically follow carriage returns.
PD_NUL	**End of Line Null Count** Indicates the number of NULL padding bytes to send after a carriage return/line-feed character.
PD_PAU	**End of Page Pause** If PD_PAU is non-zero, an auto page pause occurs upon reaching a full screen of output. See PD_PAG for setting page length.
PD_PAG	**Page Length** Contains the number of lines per screen (or page).

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_BSP	**Backspace "Input" Character** Indicates the input character recognized as backspace. See PD_BSE and PD_BSO.
PD_DEL	**Delete Line Character** Indicates the input character recognized as the delete line function. See PD_DLO.
PD_EOR	**End of Record Character** Defines the last character on each line entered (I$Read, I$ReadLn). An output line is terminated (I$WritLn) when this character is sent. Normally PD_EOR should be set to $0D. **WARNING:** If PD_EOR is set to 0, SCF's I$ReadLn ***never*** terminates unless an EOF or error occurs.
PD_EOF	**End of File Character** This field defines the end-of-file character. SCF returns an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input.
PD_RPR	**Reprint Line Character** If this character is input, SCF (I$ReadLn) reprints the current input line. A carriage return is also inserted in the input buffer for PD_DUP (see below) to make correcting typing errors more convenient.
PD_DUP	**Duplicate Last Line Character** If this character is input, SCF (I$ReadLn) duplicates whatever is in the input buffer through the first PD_EOR character. Normally, this is the previous line typed.

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_PSC	**Pause Character** If this character is typed during output, output is suspended before the next end-of-line. This also deletes any ***type ahead*** input for I$ReadLn.
PD_INT	**Keyboard Interrupt Character** If this character is input, SCF sends a keyboard interrupt signal to the last user of this path. It terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code. PD_INT is normally set to a control-C character.
PD_QUT	**Keyboard Abort Character** If this character is input, SCF sends a keyboard abort signal to the last user of this path. It terminates the current I/O request (if any) with an error code identical to the keyboard abort signal code. PD_QUT is normally set to a control-E character.
PD_BSE	**Backspace “Output” Character (Echo Character)** This field indicates the backspace character to echo when PD_BSP is input. See PD_BSP and PD_BSO.
PD_OVF	**Line Overflow Character** If I$ReadLn has satisfied its input byte count, SCF ignores any further input characters until an end-of-record character (PD_EOR) is received. It echoes the PD_OVF character for each byte ignored. PD_OVF is usually set to the terminal’s bell character.

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_PAR	**Parity Code, Number of Stop Bits, and Bits/Character** Bits zero and one indicate the parity as follows: 0 = no parity 1 = odd parity 3 = even parity Bits two and three indicate the number of bits per character as follows: 0 = 8 bits/character 1 = 7 bits/character 2 = 6 bits/character 3 = 5 bits/character Bits four and five indicate the number of stop bits as follows: 0 = 1 stop bit 1 = 1 1/2 stop bits 2 = 2 stop bits Bits six and seven are reserved.

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_BAU	**Software Adjustable Baud Rate** This one-byte field indicates the baud rate as follows: 0 = 50 baud 1 = 75 baud 2 = 110 baud 3 = 134.5 baud 4 = 150 baud 5 = 300 baud 6 = 600 baud 7 = 1200 baud 8 = 1800 baud 9 = 2000 baud A = 2400 baud B = 3600 baud C = 4800 baud D = 7200 baud E = 9600 baud F = 19200 baud 10 = 38400 baud FF = External
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 (a keyboard and video display) to be one logical device. Usually PD_D2P refers to the name of the same device descriptor in which it appears.
PD_XON	**X-ON Character** See PD_XOFF below.

Table B-5 SCF Device Descriptor Modules (continued)

Name	Description
PD_XOFF	**X-OFF Character** The X-ON and X-OFF characters are used to support software handshaking. Output from a SCF device is halted immediately when PD_XOFF is received and does not resume until PD_XON is received. This allows the distant end to control its incoming data stream. Input to a SCF device is controlled by the driver. If the input FIFO is nearly full, the driver sends PD_XOFF to the distant end to halt input. When the FIFO has been emptied sufficiently, the driver resumes input by sending the PD_XON character. This allows the driver to control its incoming data stream. **NOTE:** When software handshaking is enabled, the driver consumes the PD_XON and PD_XOFF characters itself.
PD_Tab	**Tab Character** In I$WritLn calls, SCF expands this character into spaces to make tab stops at the column intervals specified by PD_Tabs. **NOTE:** SCF does not know the effect of tab characters on particular terminals. Tab characters may expand incorrectly if they are sent directly to the terminal.
PD_Tabs	**Tab Field Size** See PD_Tab.

Note

In the following table, 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`.

Table B-6 SCF Device Descriptor Initialization Table

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Type
$49	PD_UPC	Upper Case Lock
$4A	PD_BSO	Backspace Option
$4B	PD_DLO	Delete Line Character
$4C	PD_EKO	Echo
$4D	PD_ALF	Automatic Line Feed
$4E	PD_NUL	End Of Line Null Count
$4F	PD_PAU	End Of Page Pause
$50	PD_PAG	Page Length
$51	PD_BSP	Backspace Input Character

Table B-6 SCF Device Descriptor Initialization Table (continued)

Device Descriptor Offset	Path Descriptor Label	Description
$52	PD_DEL	Delete Line Character
$53	PD_EOR	End Of Record Character
$54	PD_EOF	End Of File Character
$55	PD_RPR	Reprint Line Character
$56	PD_DUP	Duplicate Line Character
$57	PD_PSC	Pause Character
$58	PD_INT	Keyboard Interrupt Character
$59	PD_QUT	Keyboard Abort Character
$5A	PD_BSE	Backspace Output
$5B	PD_OVF	Line Overflow Character (bell)
$5C	PD_PAR	Parity Code, number of Stop Bits, and number of Bits/Character
$5D	PD_BAU	Adjustable Baud Rate
$5E	PD_D2P	Offset To Output Device Name
$60	PD_XON	X-ON Character
$61	PD_XOFF	X-OFF Character

Table B-6 SCF Device Descriptor Initialization Table (continued)

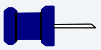
Device Descriptor Offset	Path Descriptor Label	Description
$62	PD_TAB	Tab Character
$63	PD_TABS	Tab Column Width

SCF Definitions of the Path Descriptor

The first 27 fields of the path options section (PD_OPT) of the SCF path descriptor are copied directly from the SCF device descriptor initialization table. The table is shown on the following page.

You can examine or change the fields with the I$GetStt and I$SetStt service requests or the tmode and xmode utilities.

You may disable the SCF editing functions by setting the corresponding control character value to 0. For example, if you set PD_INT to 0, there is no ***keyboard interrupt*** character.



Note

Full definitions for the fields copied from the device descriptor are available in the previous section. The additional path descriptor fields are defined below.

Table B-7 SCF Path Descriptors

Name	Description
PD_TBL	**Device Table Entry** A user-visible copy of the device table entry for the device.
PD_COL	**Current Column** The current column position of the cursor.
PD_ERR	**Most Recent Error Status** The most recent I/O error status.

Note

In the following table, 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`.

Table B-8 SCF Path Descriptor Module Option Table for I/O Editing

Offset	Name	Description
$80	PD_DTP	Device Type
$81	PD_UPC	Upper Case Lock
$82	PD_BSO	Backspace Option
$83	PD_DLO	Delete Line Character
$84	PD_EKO	Echo
$85	PD_ALF	Automatic Line Feed
$86	PD_NUL	End Of Line Null Count
$87	PD_PAU	End Of Page Pause
$88	PD_PAG	Page Length
$89	PD_BSP	Backspace Input Character
$8A	PD_DEL	Delete Line Character

Table B-8 SCF Path Descriptor Module Option Table for I/O Editing (continued)

Offset	Name	Description
$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
$90	PD_INT	Keyboard Interrupt Character
$91	PD_QUT	Keyboard Abort Character
$92	PD_BSE	Backspace Output
$93	PD_OVF	Line Overflow Character (bell)
$94	PD_PAR	Parity Code, number of Stop Bits, and number of Bits/Character
$95	PD_BAU	Adjustable Baud Rate
$96	PD_D2P	Offset To Output Device Name
$98	PD_XON	X-ON Character
$99	PD_XOFF	X-OFF Character
$9A	PD_TAB	Tab Character
$9B	PD_TABS	Tab Column Width

Table B-8 SCF Path Descriptor Module Option Table for I/O Editing (continued)

Offset	Name	Description
$9C	PD_TBL	Device Table Entry
$A0	PD_Col	Current Column
$A2	PD_Err	Most Recent Error Status
$A3		Reserved

SBF Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for SBF devices. The initialization table immediately follows the standard device descriptor module header fields (see Chapter 3: OS-9 Input/Output System for full descriptions). A graphic representation of the table is shown in **Table B-9**. The size of the table is defined in the `M$Opt` field.

Table B-9 Initialization Table for SBF Device Descriptor Module

Device Descriptor Offset	Path Descriptor Label	Description
$48	PD_DTP	Device Type
$49	PD_TDrv	Tape Drive Number
$4A	PD_SBF	Reserved
$4B	PD_NumBlk	Maximum Number of Blocks to Allocate
$4C	PD_BlkSiz	Logical Block Size
$50	PD_Prior	Driver Process Priority
$52	PD_SBFFlags	SBF Path Flags
$53	PD_DrivFlag	Driver Flags
$54	PD_DMAMode	Direct Memory Access Mode
$56	PD_ScsiID	SCSI Controller ID

Table B-9 Initialization Table for SBF Device Descriptor Module (continued)

Device Descriptor Offset	Path Descriptor Label	Description
$57	PD_ScsiLUN	LUN on SCSI Controller
$58	PD_ScsiOpts	SCSI Options Flags

Note

In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, make the following adjustment: (M$DTyp - PD_OPT).

For example, to access the tape drive number in a device descriptor, use the following value: PD_TDrv + (M$DTyp - PD_OPT). To access the tape drive number in the path descriptor, use PD_TDrv. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: sys.l or usr.l.

Table B-10 SBF Device Descriptor Modules

Name	Description
PD_DTP	**Device Class** This field is set to three for SBF devices. (0=SCF, 1=RBF, 2=PIPE, 3=SBF, 4=NET)
PD_TDrv	**Tape Drive Number** This is used to associate a one-byte integer with each drive a controller handles. • If using dedicated (for example, non-SCSI bus) controllers, this field usually defines both the ***logical*** and ***physical*** drive number of the tape drive. • If using tape drives connected to SCSI controllers, this number defines the ***logical*** number of the tape drive to the device driver. The ***physical*** controller ID and LUN are specified by the PD_ScsiID and PD_ScsiLUN fields. Each controller's drives should be numbered 0 to n-1 (n is the maximum number of drives the controller can handle). This number also defines how many drive tables are required by the driver and SBF. SBF verifies this number against SBF_NDRV prior to calling the driver.
PD_NumBlk	**Number of Buffers/Blocks Used for Buffering** Specifies the maximum number of buffers to be allocated by SBF for use by the auxiliary process in buffered I/O. If this field is set to 0, unbuffered I/O is specified.

Table B-10 SBF Device Descriptor Modules (continued)

Name	Description
PD_BlkSiz	**Logical Block Size Used for I/O** Specifies the size of the buffer for SBF to allocate. This buffer size is used when allocating multiple buffers used in buffered I/O. Unless the driver manages partial physical blocks, this size should be an integer multiple of the physical tape block size.
PD_Prior	**Driver Process Priority** The priority at which SBF's auxiliary process runs. This value is used during initialization. Changing this value after initialization has no effect.
PD_SBFFlags	**SBF Path Flags** Specifies the actions SBF takes when the path is closed. You can update this field using GetStat/SetStat (SS_Opt). SBF supports the following flag definitions: bit 0:(f_rest_b) 0 = No rewind on close. 1 = Rewind on close. bit 1:(f_offl_b) 0 = Do not put drive off-line on close. 1 = Put drive off-line on close. bit 2:(f_eras_b) 0 = Do not erase to end-of-tape on close. 1 = Erase to end-of-tape on close.

Table B-10 SBF Device Descriptor Modules (continued)

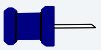
Name	Description
PD_DrivFlag	**Driver Flags** This field is available for use by the device driver. **NOTE:** References to these flags are often made using the PD_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD_SBFFlags. References to PD_DrivFlag should use a value of PD_Flags + 1.
PD_DMAMode	**Direct Memory Access Mode** This field is hardware specific. If available, you can use this word to specify the DMA Mode of the driver.
PD_ScsiID	**SCSI Controller ID** This is the ID number of the SCSI controller attached to the device. The driver uses this number when communicating with the controller.

Table B-10 SBF Device Descriptor Modules (continued)

Name	Description
PD_ScsiLUN	**Logical Unit Number of SCSI Device** This number is the value to use in the SCSI command block to identify the logical unit on the SCSI controller. This number may be different from PD_TDrv to eliminate allocation of unused drive table storage. PD_TDrv indicates the logical number of the drive to the driver and SBF (drive table to use). PD_ScsiLUN is the physical drive number on the controller.
PD_ScsiOpts	**SCSI Driver Options Flags** This field allows SCSI device options and operation modes to be specified. It is the driver's responsibility to use or reject these if applicable: bit 0: 0 = ATN not asserted (no disconnects allowed). 1 = ATN asserted (disconnects allowed). bit 1: 0 = Device cannot operate as a target. 1 = Device can operate as a target. bit 2: 0 = asynchronous data transfers. 1 = synchronous data transfers. bit 3: 0 = parity off. 1 = parity on. All other bits are reserved.

SBF Definitions of the Path Descriptor

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



Note

In the following table, offset refers to the location of a path descriptor field relative to the starting address of the path descriptor. Path 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`.

Table B-11 Path Descriptor PD_OPT for SBF

Offset	Name	Description
$80	PD_DTP	Device Type
$81	PD_TDrv	Tape Drive Number
$82	PD_SBF	Reserved
$83	PD_NumBlk	Maximum Number of Blocks to Allocate
$84	PD_BlkSiz	Logical Block Size
$88	PD_Prior	Driver Process Priority

Table B-11 Path Descriptor PD_OPT for SBF (continued)

Offset	Name	Description
$8A	PD_SBFFlags†	SBF Path Flags
$8B	PD_DrivFlag†	Driver Flags
$8C	PD_DMAMode	Direct Memory Access Mode
$8E	PD_ScsiID	SCSI Controller ID
$8F	PD_ScsiLUN	LUN on SCSI controller
$90	PD_ScsiOpts	SCSI Options Flags

† References to these flags are often made using the PD_Flags offset (defined in sys.l and usr.l). This reference is equivalent to PD_SBFFlags. References to PD_DrivFlag should use a value of PD_Flags + 1.

Pipe Device Descriptor Modules

This section describes the definitions of the initialization table contained in device descriptor modules for pipe devices. The initialization table immediately follows the standard device descriptor module header fields (see Chapter 3: OS-9 Input/Output System for full descriptions). A graphic representation of the table is shown in **Table B-12**. The size of the table is defined in the `M$Opt` field.

Note

In this table the offset values are the device descriptor offsets, while the labels are the path descriptor offsets. To correctly access these offsets in a device descriptor using the path descriptor labels, the following adjustment must be made: (`M$DTyp - PD_OPT`).

For example, to access the default buffer size in a device descriptor, use the following value: `PD_BufSz + (M$DTyp - PD_OPT)`. Module offsets are resolved in assembly code by using the names shown here and linking with the relocatable library: `sys.l` or `usr.l`.

Table B-12 Initialization Table for Pipe Device Descriptor Module

Device Descriptor Offset	Path Descriptor Label	Description
$48	`PD_DTP`	Device Type
$49		Reserved
$4A	`PD_BufSz`	Default pipe buffer size

Table B-13 Pipe Device Descriptor Modules

Name	Description
PD_DTP	**Device Class** (0=SCF 1=RBF 2=PIPE 3=SBF 4=NET) This field is set to 2 for pipe devices.
PD_BufSz	**Default Pipe Buffer Size** 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 (see Path Descriptor definitions) is used.

Pipe Definitions of the Path Descriptor

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

Note

In the following table, 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.

Table B-14 Path Descriptor PD_OPT for Pipeman

Offset	Name	Description
$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

Table B-15 Pipe Definitions of the Path Descriptor

Name	Description
DV_DTP	**Device Type** This field is set to 2 for Pipe devices. (0 = SCF, 1 = RBF, 2 = PIPE, 3 = SBF, 4 = NET)
PD_BufSz	**Default Pipe Buffer Size** 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 is 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)**

Appendix C: Error Codes

This appendix lists OS-9 error codes in numerical order. It contains the following topics:

Error Codes

OS-9 error codes are categorized as follows:

Table C-1 OS-9 Error Codes

Range	Description
`000:001 - 000:031`	**Miscellaneous Errors** For more information, refer to **Miscellaneous Errors**.
`000:032 - 000:047`	**Ultra C related Errors** For more information, refer to **Ultra C Related Errors**.
`000:064-000:067`	**Math Trap Related Errors** For more information, refer to **Math Trap Errors**.
`000:102 - 000:163`	**Processor Exception Errors** Error codes in this range are reserved to indicate a processor related exception occurred on behalf of the program. Only those listed within this range can occur on behalf of the user program. All other numbers between 100 - 163 are reserved. Unless the program provides for special handling of the exception condition (`F$STrap`), the error is fatal and the program terminates. The listed errors that fall between 100-163 represent the hardware exception vector plus 100. For more information, refer to **Processor Exception Errors**.

Table C-1 OS-9 Error Codes (continued)

Range	Description
000:164 - 000:176	**Miscellaneous Errors** For more information, refer to **Miscellaneous Errors**
000:177	**Semaphore Error** For more information, refer to **Semaphore Errors**.
000:200 - 000:239	**Operating System Errors** These errors are normally generated by the kernel or file managers. For more information, refer to **Operating System Errors**.
000:240 - 000:255	**I/O Errors** These error codes are generated by device drivers or file managers. For more information, refer to **I/O Errors**.
001:000 - 001:001	**Compiler Errors** For more information, refer to **Compiler Errors**.
006:100 - 006:206	**RAVE Errors** For more information, refer to **Rave Errors**.
007:001 - 007:029	**Internet Errors** For more information, refer to **Internet Errors**.
008:001 - 008:017	**ISDN Errors** For more information, refer to **ISDN Errors**.

Miscellaneous Errors

Table C-2 OS-9 Miscellaneous Errors

Number	Name	Description
000:001		**Process has aborted**
000:002	S$Abort signal	**Keyboard Quit** The keyboard abort signal (S$Abort) was sent. This is usually generated by typing <control>e.
000:003	S$Intrpt signal	**Keyboard Interrupt** The keyboard interrupt signal (S$Intrpt) was sent. This is usually generated by typing <control>c.
000:004	S$HangUp signal	**Modem Hang-up** The modem hang-up signal (S$HangUp) was sent. This is usually generated when the device driver detects the loss of the data carrier.

Ultra C Related Errors

Table C-3 OS-9 Ultra C Related Errors

Number	Name	Description
000:032	E_SIGABRT	**Abort Signal** An abort signal was received.
000:033	E_SIGFPE	**Erroneous Math Operation** An erroneous math operation was received.
000:034	E_SIGILL	**Illegal Function Image Signal** An illegal function image signal was received.
000:035	E_SIGSEGV	**Segment Violation** A segment violation (bus error) signal was received.
000:036	E_SIGTERM	**Termination Request Signal** A termination request signal was received.
000:037	E_SIGALRM	**Elapsed Alarm Time** An elapsed alarm time signal was received.
000:038	E_SIGPIPE	**No Readers for Pipe** A write to a pipe with no readers signal was received.

Table C-3 OS-9 Ultra C Related Errors (continued)

Number	Name	Description
000:039	E_SIGUSR1	**User Signal Number 1** A user signal number 1 was received.
000:040	E_SIGUSR2	**User Signal Number 2** A user signal number 2 was received.
000:041	E_SIGADDR	**Address Error Signal** An address error signal was received.
000:042	E_SIGCHK	**CHK Signal** A `chk` instruction signal was received.
000:043	E_SIGTRAPV	**TRAPV Signal** A `trapv` instruction signal was received.
000:044	E_SIGPRIV	**Privilege Violation Signal** A privilege violation signal was received.
000:045	E_SIGTRACE	**Trace Exception Signal** A trace exception signal was received.

Table C-3 OS-9 Ultra C Related Errors (continued)

Number	Name	Description
000:046	E_SIG1010	**Line A Exception Signal** A line-A exception signal was received.
000:047	E_SIG1111	**Line F Exception Signal** A line-F exception signal was received.

Math Trap Errors

Table C-4 OS-9 Math Trap Errors

Number	Name	Description
000:064	E$IllFnc	**Illegal Function Code** A math trap handler error.
000:065	E$FmtErr	**Format Error** A math trap handler error. An ASCII to numeric format conversion error.
000:066	E$NotNum	**Number Not Found** A math trap handler error.
000:067	E$IllArg	**Illegal Argument** A math trap handler error.

Processor Exception Errors

Table C-5 OS-9 Processor Exception Errors

Number	Name	Description
000:102	E$BusErr	**Bus Error** A bus error exception occurred.
000:103	E$AdrErr	**Address Error** An address error exception occurred.
000:104	E$IllIns	**Illegal Instruction** An illegal instruction exception occurred.
000:105	E$ZerDiv	**Zero Divide** An integer zero divide exception occurred.
000:106	E$Chk	**Check** A CHK or CHK2 instruction exception occurred.
000:107	E$TrapV	**Trap** A TRAPV, TRAPcc, or FTRAPcc instruction exception occurred.
000:108	E$Violat	**Privilege Violation** A privilege violation exception occurred.

Table C-5 OS-9 Processor Exception Errors (continued)

Number	Name	Description
000:109	E$Trace	**Uninitialized Trace Exception** An uninitialized trace exception occurred.
000:110	E$1010	**1010 Trap** An A Line emulator exception occurred.
000:111	E$1111	**1111 Trap** An F Line emulator exception occurred.
000:112	E$Resrvd	**Invalid Exception** An invalid exception occurred.
000:113	E$CProto	**Co-processor Protocol Violation**
000:114	E$StkFmt	**Format Error** A system stack frame format error occurred.
000:115	E$UnIRQ	**Uninitialized Interrupt Occurred**
000:116-000:123		**Reserved**
000:124		**Spurious Interrupt Occurred**
000:133-000:147	E$Trap	**Uninitialized User TRAP 1-15 Executed**

Table C-5 OS-9 Processor Exception Errors (continued)

Number	Name	Description
000:148	E$FPUnordC	**FPCP Error** Branch or set on floating point unordered condition error.
000:149	E$FPInxact	**FPCP Error** Floating point inexact result.
000:150	E$FPDivZer	**FPCP Error** Floating point divide by zero error.
000:151	E$FPUndrFl	**FPCP Error** Floating point underflow error.
000:152	E$FPOprErr	**FPCP Error** Floating point operand error.
000:153	E$FPOverFl	**FPCP Error** Floating point overflow error.
000:154	E$FPNotNum	**FPCP Error** Floating point NAN signaled.
000:155	E$UnData	**FPCP Error** Floating point unimplemented Data Type
000:156	E$MMUConf	**PMMU Configuration Error**
000:157	E$MMUAcces	**PMMU Illegal Operation**

Table C-5 OS-9 Processor Exception Errors (continued)

Number	Name	Description
000:158		**PMMU Access Level Violation**
000:159- 000:163		**Invalid Exception**

Miscellaneous Errors

Table C-6 OS-9 Miscellaneous Errors

Number	Name	Description
000:164	E$Permit	**No Permission** A super user must own the process or module in order to perform the requested function.
000:165	E$Differ	**Different Arguments** The arguments to F$ChkMem do not match.
000:166	E$StkOvf	**Stack Overflow** The process' system or user stack was about to overflow. This error could be caused by: F$ChkMem can cause this error if the pattern string is too complex. The data area associated with a user exception handler (F$STrap) is not part of your data area. This can be caused by specifying an invalid address or when your stack is nested too deep (unmasking signals within the signal handler can cause this). The signal handler is nested too deep. Unmasking signals within the signal handler can cause this.

Table C-6 OS-9 Miscellaneous Errors (continued)

Number	Name	Description
000:167	E$EvntID	**Illegal Event ID** You specified an invalid or illegal event ID number.
000:168	E$EvNF	**Event Name Not Found** You tried to link to or delete an event, but the name of the event is not in the event table.
000:169	E$EvBusy	**Event Busy** You tried to delete an event, but its link count is non-zero. This can also occur if you try to create a named event that already exists.
000:170	E$EvParm	**Impossible Event Parameter** You passed parameters to F$Event that are not possible.
000:171	E$Damage	**System Damage** A system data structure has been corrupted.
000:172	E$BadRev	**Incompatible Revision** The software revision is not compatible with the operating system revision.

Table C-6 OS-9 Miscellaneous Errors (continued)

Number	Name	Description
000:173	E$PthLost	**Path Lost** The path became lost. This usually occurs when: • A network node has gone down. • A serial connection has lost data carrier. • A pipe path has been broken due to an SS_Break SetStat.
000:174	E$BadPart	**Bad Partition** Bad partition data or no active partition.
000:175	E$Hardware	**Hardware Damage Has Been Detected** E$Hardware usually occurs when the driver fails to detect the correct responses from the hardware. This can occur due to hardware failure or an incorrect hardware configuration.
000:176	E$SectSize	**Invalid Sector Size** The sector size of an RBF device must be a binary multiple of 256 (256, 512, 1024, and so forth). The maximum sector size is 32768.

Semaphore Errors

Table C-7 OS-9 Semaphore Errors

Number	Name	Description
000:177	E$BSig	**Unexpected or Bad Signal**

Operating System Errors

Table C-8 OS-9 Operating System Errors

Number	Name	Description
000:200	E$PthFul	**Path Table Full** A user program has tried to open more than 32 I/O paths simultaneously. When the system path table gets full, the kernel automatically expands it. However, this error could be returned if there is not enough contiguous memory to expand the table.
000:201	E$BPNum	**Illegal Path Number** The path number was too large or for a non-existent path. This could occur whenever passing a path number to an I/O call.
000:202	E$Poll	**Interrupt Polling Table Full** You tried to install an IRQ Service Routine into the system polling table, but the table was full. To install another interrupt producing device, you must first remove one already in the table. The system's `INIT` module specifies the maximum number of IRQ devices you may install.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:203	E$BMode	**Illegal Mode** You tried to perform an I/O function that the device or file cannot perform. This could occur, for instance, when trying to read from an output file (for example, a printer).
000:204	E$DevOvf	**Device Table Full** OS-9 cannot add the specified device to the system because the device table is full. To install another device, you must first remove one already in the table. The system's `INIT` module specifies the maximum number of devices supported, but this may be changed to add more.
000:205	E$BMID	**Illegal Module Header** OS-9 cannot load the specified module because its module sync code is incorrect.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:206	E$DirFul	**Module Directory Full** The module directory is full. To load or create another module, you must first unlink one already in the directory. Although OS-9 expands the module directory when it becomes full, this error may be returned because there is not enough memory or the memory is too fragmented to use.
000:207	E$MemFul	**Memory Full** The process can not execute because there is not enough contiguous RAM free. This can also occur if a process has already been allocated the maximum number of blocks permitted by the system.
000:208	E$UnkSvc	**Unknown Service Code** The specified service call has an unknown or invalid service code number. This can also occur if a GetStat/SetStat call is made with an unknown status code.
000:209	E$ModBsy	**Module Busy** You tried to access a non-sharable module or non-sharable device that is already in use.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:210	E$BPAddr	**Boundary Error** A memory de-allocation request was not passed a valid block address or you tried to de-allocate memory not previously assigned. The system detects trouble when the buffer returns to free memory or if it is used as the destination of a data transfer, such as I$Read.
000:211	E$EOF	**End of File** An end of file condition was encountered on a read operation.
000:212	E$VctBsy	**Vector Busy** A device is trying to use an IRQ vector that another device is currently using.
000:213	E$NES	**Non-Existing Segment** A search was made for a disk file segment that cannot be found. The device may have a damaged file structure.
000:214	E$FNA	**File Not Accessible** You tried to open a file or device without the correct access permissions. Check the file's attributes and the owner ID.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:215	E$BPNam	**Bad Path Name** There is a syntax error in the specified pathlist (for example, an illegal character). This can occur whenever you reference a path by name.
000:216	E$PNNF	**Path Name Not Found** The specified pathlist cannot be found. This could be caused by misspellings or incorrect directories.
000:217	E$SLF	**Segment List Full** A file is too fragmented to be expanded any further. This can be caused by expanding a file many times without regard to memory allocation. It also occurs on disks with little free memory or disks whose free memory is too scattered. A simple way to solve this problem is to copy the file (or disk). This should move it into contiguous areas.
000:218	E$CEF	**File Already Exists** You tried to create a file using a name that already appears in the current directory.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:219	E$IBA	**Illegal Block Address** A search for an illegal block address has occurred. An invalid pointer or block size has been passed or the device's file structure is damaged.
000:220	E$HangUp	**Telephone (Modem) Data Carrier Lost** You tried to perform an I/O operation on a path after irrecoverable line problems occurred (for example, data carrier lost). It may be returned from network devices if the network connection is lost.
000:221	E$MNF	**Module Not Found** A request is made to link to a module that is not found in the module directory. Modules whose headers have been modified or corrupted can not be found.
000:222	E$NoClk	**No Clock** This error returns when a request is made that uses the system clock and the system has no clock running. For example, a timed sleep request returns this error if there is no system clock running. Use `setime` to start the system clock.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:223	E$DelSP	**Suicide Attempt** A user requested de-allocation and return of the memory where the user's stack is located. This could be caused, for example, by using the F$Mem system call to contract the data memory of the specified process.
000:224	E$IPrcID	**Illegal Process Number** A system call was passed a process ID to a non-existent process or a process you may not access.
000:225	E$Param	**Bad Parameter** You passed a service request an illegal or impossible parameter.
000:226	E$NoChld	**No Children** You made an F$Wait request, but your process has no child process for which to wait.
000:227	E$ITrap	**Illegal Trap Code** You used an unavailable (already in use) or invalid trap code in a F$TLink call.
000:228	E$PrcAbt	**Process Aborted** A process is aborted by the kill signal code.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:229	E$PrcFul	**Process Table Full** The system process table is full (too many processes currently running). Although OS-9 automatically tries to expand the table, this error may occur if there is not enough contiguous memory to do so.
000:230	E$IForkP	**Illegal Parameter Area** Ridiculous parameters were passed to a fork call.
000:231	E$KwnMod	**Known Module** You tried to install a module that is already in memory.
000:232	E$BMCRC	**Incorrect Module CRC** The specified module being checked or verified has a bad CRC value. Use `fixmod` to generate a valid CRC.
000:233	E$USigP	**Unprocessed Signal Pending**
000:234	E$NEMod	**Non-Executable Module** A process tried to execute a module with a type other than program/object.
000:235	E$BNam	**Bad Name** There is a syntax error in the specified name.

Table C-8 OS-9 Operating System Errors (continued)

Number	Name	Description
000:236	E$BMHP	**Bad Parity** The specified module has bad module header parity.
000:237	E$NoRAM	**RAM Full** 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. **NOTE:** F$Mem is no longer available. Use F$SRqMem instead.
000:238	E$DNE	**Directory Not Empty** You tried to remove the directory attribute from a directory that is not empty.
000:239	E$NoTask	**No Task Number Available** All task numbers are currently in use and a request was made to execute or create a new task. This might be returned by an OS-9 System Security Module (SSM).

I/O Errors

Table C-9 OS-9 I/O Errors

Number	Name	Description
000:240	E$Unit	**Illegal Drive Number**
000:241	E$Sect	**Bad Sector** Bad disk sector number.
000:242	E$WP	**Write Protect** Media is write protected.
000:243	E$CRC	**CRC Error** A CRC error occurred on read or write verify.
000:244	E$Read	**Read Error** A data transfer error occurred during disk read operation, or SCF (terminal) input buffer overrun.
000:245	E$Write	**Write Error** A hardware error occurred during disk write operation.
000:246	E$NotRdy	**Not Ready** Device has not ready status.
000:247	E$Seek	**Seek Error** You tried to perform a physical seek to a non-existent sector.

Table C-9 OS-9 I/O Errors (continued)

Number	Name	Description
000:248	E$Full	**Media Full** Insufficient free space on media.
000:249	E$BTyp	**Wrong Type** You tried to read incompatible media (you attempted to read a double-sided disk on single-sided drive).
000:250	E$DevBsy	**Device Busy** A non-sharable device is in use.
000:251	E$DIDC	**Disk ID Change** 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. Therefore, it is possible to swap disks without RBF noticing. This check helps to prevent this possibility.

Table C-9 OS-9 I/O Errors (continued)

Number	Name	Description
000:252	E$Lock	**Record Is Locked-Out** Another process is accessing the requested record. Normal record locking routines wait forever for a record in use by another user to become available. However, RBF may be told to wait for a finite amount of time with a SetStat. If the time expires before the record becomes free, this error is returned.
000:253	E$Share	**Non-Sharable File Busy** The requested file or device has the single user bit set or it 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 currently open file.

Table C-9 OS-9 I/O Errors (continued)

Number	Name	Description
000:254	E$DeadLk	**I/O Deadlock** Two processes are trying to use the same two disk areas simultaneously. Each process is locking out the other process, producing the I/O deadlock. One of the processes must release its control to allow the other to proceed.
000:255	E$Format	**Device Is Format Protected** You tried to format a format protected disk. To allow the device to be formatted, change a bit in the device descriptor. Formatting is usually inhibited on hard disks to prevent erasure.

Compiler Errors

Table C-10 OS-9 Compiler Errors

Number	Name	Description
001:000	ERANGE	**ANSI C Number Out of Range**
001:001	EDOM	**ANSI C Number Not in Domain**

Rave Errors

Table C-11 OS-9 Rave Errors

Number	Name	Description
006:000	E$IllPrm	**Illegal Parameter** An illegal parameter was passed to a function.
006:001	E$IdFull	**Identifier Table Full** An ID table could not be expanded any further.
006:002	E$BadSiz	**Bad Size Error**
006:003	E$RgFull	**Region definition full (overflow)** The region is too complex.
006:004	E$UnID	**Unallocated Identifier Number** An attempt was made to use an ID number for an object (drawmap, action region, etc.) that was not allocated.
006:005	E$NullRg	**Null Region** You attempted to use a null region.
006:006	E$BadMod	**Bad Drawmap/Pattern Mode** An illegal mode was passed to create a drawmap or pattern.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:007	E$NoFont	**No Active Font** No font was activated when an attempt to output text was made.
006:008	E$NoDM	**No Drawmap** No character output drawmap was available when attempting an `_os_write` or `_os_writeln`.
006:009	E$NoPlay	**No Audio Play in Progress** An attempt was made to stop an audio play when none was in progress.
006:010	E$Abort	**Audio Record/Play Aborted** An `sm_off()` call was done to abort a play in progress.
006:011	E$QFull	**Audio Queue is Full** The driver queue could not handle the number of soundmaps you were attempting to output.
006:012	E$Busy	**Audio Processor Is Busy**
006:100	E_RES_NOSLOT	**No Free Slot Is Left in the Resource Table** `res_load()` returns this error when no more resource modules can be loaded for this application.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:101	E_RES_BADSLOT	**Specified Resource Module ID Is Not a Valid Slot in the Resource Table** `res_set()` and `res_free()` return this error if the resource module ID does not correspond to a valid resource module.
006:102	E_RES_NOSHARE	**Resource Is Not Sharable** `res_get()` returns this error if the resource is non-sharable.
006:103	E_RES_NOTYPE	**Resource Type Is Bad** The specified type was not found in the resource map.
006:104	E_RES_NORES	**Bad Resource ID** The specified resource ID was not found in the resource map.
006:110	E_REQ_NOITEMS	**No Items Are Specified for Request** `req_create()` returns this error if the `rq_numits` field is `0`.
006:111	E_REQ_BADITEM	**Out of Range Item Number** A specified item number is greater than the number of items in a request.
006:112	E_REQ_BADCOLS	**Out of Range Column Number** `req_create()` returns this error if the `rq_numcols` field is `0`.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:113	E_REQ_BADPTR	**Bad Item Array Pointer** `req_create()` returns this error if the `rq_choices` pointer is not initialized correctly.
006:114	E_REQ_NOCREATE	**Could Not Create Request** `req_create()` returns this error if the request could not be created. Other functions may return it if you try to use a non-created request.
006:115	E_REQ_TIMEOUT	**Modal Request Has Timed Out** `req_make()` returns this error if the time-out value is reached before you make a selection in the request.
006:116	E_REQ_NOSEL	**No Selection Was Made for a Modal Request** `req_make()` returns this error if you clicked outside of the request or on a disabled item, thus making no selection.
006:117	E_REQ_DEFID	**Bad Definition Function ID** You did not correctly initialize the `rq_type` field.
006:118	E_REQ_DEFACT	**Bad Definition Action Code** You called the definition function with an invalid `action` parameter.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:119	E_REQ_STATE	**Bad Item State Value** You called the REQ_A_ISTATE action of the standard definition function with an invalid item state value.
006:120	E_REQ_BADRECT	**Bad Request Rectangle** req_create() returns this error if the rq_rect structure is not initialized correctly.
006:130	E_CNT_BHVID	**Bad Standard Behavior ID** The cnt_bhv field was not correctly initialized, or a bad **bhv** parameter was passed to cntl_bhv().
006:131	E_CNT_DEFID	**Bad Standard Definition ID** The cnt_def field was not correctly initialized, or a bad def parameter was passed to cntl_def().
006:132	E_CNT_DEFACT	**Bad Action for Definition Function** You passed a bad action parameter to cntl_def().
006:133	E_CNT_BHVACT	**Bad Action for Behavior Function** You passed a bad action parameter to cntl_bhv().

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:134	E_CNT_STATE	**Bad Control State** A bad `state` parameter was passed to `cntl_state()`, or a bad value was found in the `cnt_state` field.
006:135	E_CNT_PART	**Bad Control Part Code** The definition function of standard controls returns this error if you try to draw a bad part of a control.
006:136	E_CNT_FLAGS	**Bad Flags** `cntl_create()` returns this error if the `cnt_flags` field is not correctly initialized.
006:137	E_CNT_MINMAX	**Bad Minimum, Maximum, or Value** `cntl_create()` returns this error if the `cnt_min`, `cnt_max`, and `cnt_value` fields are not correctly ordered. The order must be: `cnt_min <= cnt_value <= cnt_max`.
006:138	E_CNT_TYPE	**Bad Control Type**

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:140	E_CLIP_DEV	**Cannot Find the Clipboard Device in Preferences** You tried to load or save the clipboard from a writable media and the device name was not found in the preferences.
006:141	E_CLIP_FULL	**Clipboard Is Full** You tried to write too many types in the clipboard.
006:142	E_CLIP_TYPE	**Type Not Represented in Clipboard** `clip_getptr()` returns this error when the requested type is not in the clipboard.
006:143	E_CLIP_ACC	**Clipboard Not Opened for Requested Access** You tried to write the clipboard which was opened for read access.
006:144	E_CLIP_CNT	**Type Offset Is Greater than Type Count** `clip_type()` returns this error if the type index is out of range of the available types.
006:145	E_CLIP_OPEN	**Clipboard Is Currently Not Open** You tried to access the clipboard before opening it.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:146	E_CLIP_INIT	**Clipboard Is Not Initialized** The clipboard was accessed before the program linked to it.
006:147	E_CLIP_CLOSE	**Clipboard Is Currently Not Closed** `clip_unlink()` returns this error if the clipboard was not closed before trying to unlink it.
006:148	E_CLIP_RW	**Cannot Rewrite, the Type Is Not in the Clipboard** `clip_rewrite()` returns this error if the type was not previously allocated with the `clip_write()` function.
006:150	E_HNDLR_UNKNOWN	**The Handler Is Unknown** `hndlr_delete()` returns this error if the handler specified is not known.
006:155	E_ATABL_NOENTRY	**No Entry Found** `atbl_delassoc()` returns this error when you try to delete an association that does not exist.
006:160	E_BOX_TABLE	**Line Table Overflow** The specified linetab array is too small to store the offsets of each line start.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:161	E_BOX_COUNT	**Text Too Long (Maximum Is 65535)** The maximum number of characters is 65535. This restriction is because line start offsets are coded on short integers.
006:162	E_BOX_TYPE	**Bad Type or Type Not Implemented**
006:163	E_BOX_MAXL	**Attempt to Draw a Line Too Long** The maximum number of characters allowed in a line is 1023.
006:164	E_BOX_NOTAB	**Need a Line Table** If BOX_F_USETAB is set in the `type` parameter, you must specify the `linetab` and `size` parameters.
006:165	E_BOX_NOFONT	**Font Not Set in Drawmap** The appropriate font(s) should have been activated in the output drawmap.
006:166	E_BOX_RECT	**Bad Rectangle** The specified rectangle is incorrect.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:180	E_INIT_VARERROR	**Global Variable Error** You passed an illegal global variable ID to `gsl_get()` or `gsl_set()`.
006:185	E_INTER_NOMOD	**No Preference Module** No preference module was linked before using any `intl_xxx()` function.
006:186	E_INTER_ILLARG	**Illegal Argument** You passed a bad argument to an internationalization function.
006:190	E_OVL_BADRECT	**Bad Rectangle for Overlay** The rectangle passed is illegal. For example, `ex <= sx` or `ey <= sy`.
006:191	E_OVL_NOTTOP	**Overlay Is Not the Top of the Stack** `ovl_close()` returns this error if you try to close an overlay which is not the top of the overlay stack.
006:192	E_OVL_UNKNOWN	**Unknown Overlay** You tried to access an overlay which was not created.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:200	E_IND_DEFID	**Bad Definition ID** The `ind_type` field was not correctly initialized, or a bad `def` parameter was passed to `ind_def()`.
006:201	E_IND_DEFACT	**Bad Definition Action** You passed a bad `action` parameter to a definition function.
006:202	E_IND_MINMAX	**Bad Minimum, Maximum, or Value** The `ind_min` or `ind_max` values were not initialized correctly or you tried to set the value of the indicator out of the minimum/maximum range.
006:203	E_IND_BADCOORDS	**Bad Coordinates** `ind_create()` returns this error if the coordinates given for the indicator are not valid.
006:204	E_IND_NOCREATE	**Indicator Not Created** You tried to use an indicator which was not properly created via `ind_create()`.

Table C-11 OS-9 Rave Errors (continued)

Number	Name	Description
006:205	E_IND_BADFLAGS	**Bad Flags** `ind_create()` returns this error when two or more mutually exclusive flags are set.
006:206	E_IND_BADPTR	**Bad Pointer** A bad (`NULL`) pointer was found where an initialized pointer should be found.

Internet Errors

Table C-12 OS-9 Internet Errors

Number	Name	Description
007:001	EWOULDBLOCK -or-	**I/O Operation Would Block** You tried to perform an operation causing a process to block on a socket in non-blocking mode.
	E_IFF_RONLY	**Read-only Path** This path is read-only.
007:002	EINPROGRESS -or-	**I/O Operation Now in Progress** You tried to perform an operation that takes a long time to complete (such as `connect()`) on a socket in non-blocking mode.
	E_IFF_WONLY	**Write-only Path** This path is write-only.
007:003	EALREADY -or-	**Operation Already in Progress** An operation was attempted on a non-blocking object that already had an operation in progress.
	E_IFF_ACTFORM	**No Active Form** No form is active.
007:004	EDESTADDRREQ -or-	**Destination Address Required** The attempted socket operation requires a destination address.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
	E_IFF_READER	**Wrong Reader** Wrong reader for this type of form.
007:005	EMSGSIZE -or-	**Message Too Long** A message sent on a socket was larger than the internal message buffer or some other network limit. Messages should be smaller than 32768 bytes.
	E_IFF_NOTIFF	**Not IFF** Not an IFF file.
007:006	EPROTOTYPE -or-	**Protocol Wrong Type for Socket** A protocol was specified that does not support the semantics of the socket type requested. For example, an AF_INET UDP protocol as SOCK_STREAM is the wrong protocol type for the socket.
	E_IFF_BADPARAM	**Bad Parameters**
007:007	ENOPROTOOPT -or-	**Bad Protocol Option** You specified a bad option or level in getsockopt() or setsockopt().
	E_IFF_BADCAT	**Bad CAT ID** Bad CAT ID for iff_open.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
007:008	EPROTONOSUPPORT -or-	**Protocol Not Supported** The requested protocol is not available or not configured for use.
	E_IFF_SIZE_UNKNOWN	**Size Unknown** Cannot skip because size is unknown.
007:009	ESOCKNOSUPPORT -or-	**Socket Type Not Supported** The requested socket type is not supported or not configured for use.
	E_IFF_NOT_DATA	**Not To the Data Yet**
007:010	EOPNOTSUPPORT -or-	**Operation Not Supported on Socket** For example, `accept()` on a datagram socket.
	E_IFF_PIPE_SEEK	**Bad Seek** You attempted to ***seek back*** in a pipe.
007:011	EPFNOSUPP -or-	**Protocol Family Not Supported** The requested protocol family is not currently supported. For example, PF_SNA.
	E_IFF_BADCHUNKSIZE	**Bad Chunk Size** The fixed size chunk is not the correct size. The reader version may be invalid.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
007:012	EAFNOSUPPORT -or-	**Protocol Does Not Support This Address Family** The current protocol does not support the requested address family. For example, PF_INET and PF_UNIX.
	E_IFF_FPNOTNUM	**No Floating Point Conversion** Cannot make floating point conversion.
007:013	EADDRINUSE	**Address Already in Use** Only one use of each address is normally permitted. Wildcard use and connectionless communication are the exceptions.
007:014	EADDRNOTAVAIL	**Cannot Assign Requested Address** Normally results when you try to create a socket with an address not on this machine.
007:015	ENETDOWN	**Network Is Down** The network hardware is not accessible.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
007:016	ENETUNREACH	**Network Is Not Reachable** The network is unreachable. Usually caused by network interface hardware that is operational, but not physically connected to the network. Or, the network has no way to reach the destination address.
007:017	ENETRESET	**Network Dropped Connection on Reset** The host you were connected to crashed and rebooted.
007:018	ECONNABORTED	**Software Caused Connection Abort** The local (host) machine caused a connection abort.
007:019	ECONNRESET	**Peer Reset Connection** A peer forcibly closed a connection. This normally results from a loss of the connection on the remote socket due to a time out or reboot.
007:020	ENOBUFS	**No Buffer Space Available** A socket operation could not be performed because the system lacked sufficient buffer space or a queue was full.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
007:021	EISCONN	**Socket Is Already Connected** You tried to connect an already connected socket. Also caused by a `sendto()` request on a connected socket to an already connected destination.
007:022	ENOTCONN	**Socket Is Not Connected** A request to send or receive data was rejected because the socket was not connected or no destination was given with a datagram socket.
007:023	ESHUTDOWN	**Cannot Send After Socket Shutdown** Socket was shut down, so additional data transmissions are not allowed.
007:024	ETOOMANYREFS	**Too Many References**
007:025	ETIMEDOUT	**Connection Timed Out** A `connect()` or `send()` request failed because the connected peer did not properly respond after a period of time. The time out period depends on the protocol used.

Table C-12 OS-9 Internet Errors (continued)

Number	Name	Description
007:026	ECONNREFUSED	**Target Refused Connection** No connection could be established because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the target host.
007:027	EBUFTOOSMALL	**Buffer Too Small for F$Mbuf Operation** The specified buffer cannot be used to support F$MBuf (SysMbuf).
007:028	ESMODEXISTS	**Socket Module Already Attached** You tried to attach a socket module when it was already successfully attached.
007:029	ENOTSOCK	**Path Is Not a Socket** You tried to perform a socket function on a path which is not a socket.

ISDN Errors

Table C-13 OS-9 ISDN Errors

Number	Name	Description
008:001	E$LnkDwn	**Line Down or Layer 1 Error on Attach** Check to make sure the line is plugged in or check to see if there was a hardware failure.
008:002	E$Conn	**Connection Error - Connection Not Made** An error occurred when the connection to the call was trying to be made. Possibilities include maximum number of connections have already been made or the connection requested is not possible.
008:003	E$RxThread	**Receive Thread Incoming Packet Handler Error** There may be something in the received packet that the receive thread process cannot identify. It may be due to a non-existent SAPI or TEI. Or the driver has sent an incorrect receive packet to the receive thread.

Table C-13 OS-9 ISDN Errors (continued)

Number	Name	Description
008:004	E$ME	**Management Entity Error** There is an error with the management entity. This could be due to a packet that the management entity software cannot interpret or requests to the management entity are not understood or cannot be carried out.
008:005	E$SAPI	**Unrecognized Service Access Point Identifier (SAPI)** The SAPI of the virtual unit or incoming message from the network is not a legal value.
008:006	E$TEI	**Terminal Endpoint Identifier (TEI) Error** The TEI of the virtual unit, TEI list, or incoming message from the network is not valid.
008:007	E$Max_TEI	**Maximum Number of Terminal Endpoints in Use** You tried to open a new virtual unit, but the number of TEIs currently open is already at the maximum. The network places this restriction on the system of eight TEIs for multipoint lines.

Table C-13 OS-9 ISDN Errors (continued)

Number	Name	Description
008:008	E$TState	**Illegal Layer 2 State** The number of the TEI state in the virtual unit is not a legal TEI state at layer 2. Valid TEI states are 1-8. Any other states are illegal.
008:009	E$TEI_Denied	**Terminal Endpoint (TEI) Initialization Denied** The network has denied the TEI request made by the file manager when opening a path to the device.
008:010	E$Primitive	**Unrecognized Primitive** A primitive has been sent to the layer 3 part of the file manager that is not understood or is an incorrect primitive for the state the call is currently in for layer 3.
008:011	E$L2In	**Layer 2 Error on Incoming Message** A message has come into the layer 2 part of the file manager that cannot be interpreted. This may be due to a message that is not understandable, or a message that cannot be acted on due to the state of the layer 2 state machine.

Table C-13 OS-9 ISDN Errors (continued)

Number	Name	Description
008:012	E$Peer_Busy	**Peer Receiver (Far End) Busy Condition** The far end has indicated it is busy and you should not send any more data until the far end can take more. This is analogous to X-ON/X-OFF protocol in serial communications.
008:013	E$K	**Maximum Number of Outstanding Messages Exceeded** The maximum number of outstanding frames (K) have been sent without the far end's acknowledgment. For the D channel and other reliable data links, this error indicates you should re-initialize layer 2.
008:014	E$MaxCRef	**Maximum Number of Call References in Use** The maximum number of call references are already in use. The path must wait until a call hangs up before trying again.
008:015	E$CRef	**Call Reference Does Not Exist** This error usually occurs when the path references a non-existent call.

Table C-13 OS-9 ISDN Errors (continued)

Number	Name	Description
008:016	E$CallProg	**Call Progress State Error** An error occurred with the call progress state machine of layer 3. Either an inappropriate action for the state is requested, or the call progress number does not correspond to a legal layer 3 state.
008:017	E$Rcvr	**Receiver Assignment/Removal Error** An error occurred when you tried to assign or remove a path from being a receiver. This error usually occurs when trying to assign when there is already a path with the same layer 3 parameters assigned on the same virtual unit.

Appendix D: OS-9 for 68K System Calls

System Calls

System calls allow you to communicate between the OS-9 operating system and assembly language level programs. There are three general categories of system calls:

- User-state
- I/O
- System-state

All system calls have a mnemonic name for easy reference:

- User and system state functions start with F$
- I/O related functions begin with I$

The mnemonic names are defined in the relocatable library file `usr.l` or `sys.l`. You should link these files with your programs.

The OS-9 I/O system calls are simple to use because the calling program does not have to allocate and set up file control blocks, sector buffers, etc. Instead, OS-9 returns a path number word when you open a file/device. You can use this path number 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; you never have to deal with them.

System-state system calls are privileged and can only execute while OS-9 is in system state (when it is processing another service request, executing a file manager, device driver, etc.). System state functions are included in this manual primarily for the benefit of those programmers who are writing device drivers and other system-level applications.

For More Information

For a full description of system state and its uses, refer to **Chapter 2: The Kernel**.

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) are returned in the MPU registers after OS-9 has processed the service request.

All system calls use a standard convention for reporting errors; if an error occurred, the carry bit of the condition code register is set and register `d1.w` contains 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 in the following pages 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 ***SEE ALSO*** listing for each service request shows related service requests and/or documentation that may yield more information about the request.

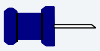
In the following system call descriptions, registers not explicitly specified as input or output parameters are not altered. Strings passed as parameters are normally terminated by a null byte.

System Calls and the System Environment

The availability of some of the system calls described in these sections varies according to the kernel environment you are using (the development kernel or the atomic kernel). In addition, some of these system calls are implemented in modules other than the kernel (for example, the I$ calls are implemented by IOMan).

The system call descriptions in the following pages apply to a development kernel environment. When system call functionality differs between development and atomic environments, the differences are noted in the description. System calls implemented by modules other than the kernel are also noted.

The following tables list each system call, the module (the kernel, IOMan, SSM, or SysCache) implementing it, and points out calls with development/atomic environment differences.



Note

F$DExec, F$DExit, and F$DFork are not available under the atomic kernel.

There are functional differences between the development and atomic kernel for F$SysID.

Table D-1 Kernel System Calls

F$Alarm (User-State)	F$AllPD	F$AllPrc	F$AProc
F$Chain	F$CmpNam	F$CpyMem	F$CRC
F$DatMod	F$DExec	F$DExit	F$DFork
F$DelPrc	F$Exit	F$Event	F$FindPD

Table D-1 Kernel System Calls (continued)

F$FIRQ	F$Fork	F$FModul	F$GBlkMp
F$GModDr	F$GPrDBT	F$GPrDsc	F$GProcP
F$Gregor	F$Icpt	F$Icpt	F$IRQ
F$Julian	F$Link	F$Move	F$Mem
F$NProc	F$PrsNam	F$RetPD	F$RTE
F$Sema	F$Send	F$SetCRC	F$SetSys
F$Sigmask	F$SigReset	F$Sleep	F$SPrior
F$SRqMem	F$SRqCMem	F$SRtMem	F$SSvc
F$STrap	F$STime	F$SUser	F$SysDbg
F$SysID	F$Time	F$TLink	F$Trans
F$UnLink	F$UnLoad	F$Wait	F$UAcct
F$VModul			

Table D-2 IOMan System Calls

F$AllBit	F$DelBit	F$IOQu	F$IODel
F$Load	F$PErr	F$SchBit	I$Attach
I$Create	I$ChgDir	I$Close	I$Delete
I$Detach	I$Dup	I$GetStt	I$MakDir

Table D-2 IOMan System Calls (continued)

I$Open	I$Read	I$ReadLn	I$Seek
I$SetStt	I$SGetSt	I$Write	I$WritLn

Table D-3 SSM System Calls

F$AllTsk	F$ChkMem	F$DelTsk	F$GSPUMp
F$Permit (User-State)	F$Protect (User-State)		

Table D-4 SysCache System Calls

F$CCtl

F$Alarm (System-State) — Set Alarm Clock

ASM Call

```
OS9 F$Alarm
```

Input

```
d0.l = Alarm ID (or zero)
d1.w = Function code
d2.l = Reserved, must be zero
d3.l = Time interval (or time)
d4.l = Date (when using absolute time)
(a0) = Register image
```

Output

```
d0.l = Alarm ID
```

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

When called from system state, `F$Alarm` executes a system-state subroutine at a specified time. It is provided for such functions as turning off a disk drive motor if the disk is not accessed for a period of time.

The register image pointed to by register `(a0)` contains an image of the registers to be passed to the alarm subroutine. The subroutine entry point must be placed in `R$pc(a0)`. The register image is copied by the `F$Alarm` request into another buffer area and may be re-used immediately for other purposes.

You can use the returned alarm ID to delete an alarm request.

The time interval is the number of system clock ticks (or 256ths of a second) to wait before the alarm subroutine is executed. If the high order bit is set, the low 31 bits are interpreted as 256ths of a second.

Note

All times are rounded up to the nearest clock tick.

The system automatically deletes a process' pending alarms when the process dies.

The alarm function code is used to select one of the related alarm functions. Not all input parameters are always needed; each function is described in the following pages.

System-state alarm subroutines must conform to the following conventions:

Input

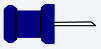
```
d0-d7 = caller's registers (R$d0-R$d7(a5))
(a0)-(a3) = caller's registers (R$a0-R$a3(a5))
(a4)  = system process descriptor pointer*
(a5)  = ptr to register image
(a6)  = system global storage pointer
```

Output

```
cc = carry set
d1.w = error code if error
```

Note

The user number in the system process descriptor is temporarily changed to the user number of original F$Alarm request. You do not have to preserve the registers d0-d7 and (a0)-(a3).



Note

The system process executes system-state alarms at priority 65535. They may never perform any function resulting in any kind of queuing, such as F$Sleep, F$Wait, F$Load, F$Event(Ev$Wait), F$IOQu, or F$Fork. When such functions are required, the caller must provide a separate process to perform the function, rather than an alarm.

WARNING

If an alarm execution routine suffers any kind of bus trap, address trap, or other hardware-related error, the system crashes.

The following F$Alarm system-state function codes are supported:

Table 7-4 F$Alarm Function Codes

Name	Description
A$AtDate (System-State)	Execute a subroutine at a Gregorian date/time.
A$AtJul (System-State)	Execute a subroutine at Julian date/time.

Table 7-4 F$Alarm Function Codes (continued)

Name	Description
A$Cycle (System-State)	Execute a subroutine at specified time intervals.
A$Delete (System-State)	Remove a pending alarm request.
A$Set (System-State)	Execute a subroutine after a specified time interval.

Possible Errors

E$BPAddr

E$MemFul

E$NoRAM

E$Param

E$UnkSvc

See Also

F$Alarm (User-State)

F$Alarm (User-State) — Set Alarm Clock

ASM Call

```
OS9 F$Alarm
```

Input

```
d0.l = Alarm ID (or zero)
d1.w = Alarm function code
d2.l = Signal code
d3.l = Time interval (or time)
d4.l = Date (when using absolute time)
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$Alarm creates an asynchronous software alarm clock timer. The timer sends a signal to the calling process when the specified time period has elapsed. A process may have multiple alarm requests pending.

The time interval is the number of system clock ticks (or 256ths of a second) to wait before an alarm signal is sent. If the high order bit is set, the low 31 bits are interpreted as 256ths of a second.

Note

All times are rounded up to the nearest clock tick.

The system automatically deletes a process' pending alarm when the process dies.

The alarm function code selects one of the several related alarm functions. Not all input parameters are always needed; each function is described in detail in the following pages.

OS-9 supports the following user-state alarm function codes:

Table D-5 Alarm Function Codes

Name	Description
A$AtDate (User-State)	Send a signal at Gregorian date/time.
A$AtJul (User-State)	Send a signal at Julian date/time.
A$Cycle (User-State)	Send a signal at specified time intervals.
A$Cycle (User-State)	Remove a pending alarm request.
A$Set (User-State)	Send a signal after specified time interval.

Possible Errors

E$BPAddr

E$MemFul

E$NoRAM

E$Param
E$UnkSvc

See Also

F$Alarm (System-State)

A$AtDate (System-State)

Execute System-State Subroutine at Gregorian Date/Time

Input

```
d0.l = Reserved, must be zero
d1.w = A$AtDate function code
d2.l = Reserved, must be zero
d3.l = Time (00hhmmss)
d4.l = Date (YYYYMMDD)
(a0) = Register image
```

Output

```
d0.l = alarm ID
```

Error Output

```
cc = carry bit set
d1.w = appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

A$AtDate executes a system-state subroutine at a specific date and time.

Note

A$AtDate only allows you to specify time to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm subroutine executes anytime the system date/time becomes greater than or equal to the alarm time.

A$AtDate (User-State) — Send Signal at Gregorian Date/Time

Input

```
d0.l = Reserved, must be zero
d1.w = A$AtDate function code
d2.l = Signal code
d3.l = Time (00hhmmss)
d4.l = Date (YYYYMMDD)
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

A$AtDate sends a signal to the caller at a specific date and time.

Note

A$AtDate only allows you to specify time to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm signal is sent anytime the system date/time becomes greater than or equal to the alarm time.

A$AtJul (System-State) Execute System-State Subroutine at Julian Date/Time

Input

```
d0.l = Reserved, must be zero
d1.w = A$AtDate or A$AtJul function code
d2.l = Reserved, must be zero
d3.l = Time (seconds after midnight)
d4.l = Date (Julian day number)
(a0) = Register image
```

Output

```
d0.l = alarm ID
```

Error Output

```
cc = carry bit set
d1.w = appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

A$AtJul executes a system-state subroutine at a specific Julian date and time.

Note

A$AtJul function only allows time to be specified to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm subroutine is executed anytime the system date/time becomes greater than or equal to the alarm time.

A$AtJul (User-State) — Send Signal at Julian Date/Time

Input

```
d0.l = Reserved, must be zero
d1.w = A$AtDate or A$AtJul function code
d2.l = Signal code
d3.l = Time (seconds after midnight)
d4.l = Date (Julian day number)
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

A$AtJul sends a signal to the caller at a specific Julian date and time.

Note

A$AtJul only allows you to specify time to the nearest second. However, it does adjust if the system's date and time have changed (via F$STime). The alarm signal is sent anytime the system date/time becomes greater than or equal to the alarm time.

A$Cycle (System-State) Execute System-State Subroutine Every N Ticks/Seconds

Input

```
d0.l = reserved, must be zero
d1.w = A$Cycle function code
d2.l = reserved, must be zero
d3.l = time interval
```

Output

```
d0.l = alarm ID
```

Error Output

```
cc = carry bit set
d1.w = appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

The cycle function is similar to the set function, except the alarm is reset after it is sent. This causes periodic execution of a system-state subroutine.

Keep cyclic system-state alarms as fast as possible and schedule them with as long a cycle as possible to avoid consuming a large portion of available CPU time.

A$Cycle (User-State) — Send Signal Every N Ticks/Seconds

Input

```
d0.l = reserved, must be zero
d1.w = A$Cycle function code
d3.l = time interval (N)
(a0) = Register Image
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

A$Cycle sends one signal after the specified time interval has elapsed. A$Cycle is similar to the A$Set (User-State) function, except the alarm is reset after it is sent, to provide a recurring periodic signal.

A$Delete (System-State) — Remove Pending Alarm Request

Input

```
d0.l = Alarm ID (or zero)
d1.w = A$Delete function code
```

Output

None

State

☐ User ☒ System ☐ I/O

Description

`A$Delete` removes a cyclic alarm or any unexpired alarm. If `0` is passed as the alarm ID, all pending alarm requests for the current process are removed.

A$Delete (User-State) — Remove Pending Alarm Request

Input

```
d0.l = Alarm ID (or zero)
d1.w = A$Delete function code
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

A$Delete removes a cyclic alarm, or any unexpired alarm. If 0 is passed as the alarm ID, all pending alarm requests are removed.

Note

If you are running a cyclic alarm, you cannot delete the alarm from within the context of the alarm itself.

A$Set (System-State)

Execute System-State Subroutine After Specified Time Interval

Input

```
d0.l = Reserved, must be zero
d1.w = A$Set function code
d2.w = Reserved, must be zero
d3.l = Time Interval
(a0) = Register image
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set to one
d1.w = Error code
```

State

☐ User ☒ System ☐ I/O

Description

A$Set executes a system-state subroutine after the specified time interval has elapsed. The time interval may be specified in system clock ticks, or 256ths of a second. The minimum time interval allowed is two system clock ticks.

A$Set (User-State) — Send Signal After Specified Time Interval

Input

```
d0.l = Reserved, must be zero
d1.w = A$Set function code
d2.w = Signal code
d3.l = Time Interval
```

Output

```
d0.l = Alarm ID
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

A$Set sends one signal after the specified time interval has elapsed. The time interval may be specified in system clock ticks, or 256ths of a second.

F$AllBit — Set Bits in Allocation Bit Map

ASM 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
```

State

☒ User ☐ System ☐ I/O

Output

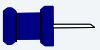
None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

Description

F$AllBit sets bits in the allocation map that were found by F$SchBit, and are now allocated. Bit numbers range from 0 to n-1, where n is the number of bits in the allocation bit map.



Note

The IOMan module implements F$AllBit.

In some applications you must allocate and de-allocate segments of a fixed resource, such as memory. One convenient way is to set up a map describing 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$AllBit, F$DelBit, and F$SchBit system calls perform the elementary bitmap operations of:

- Finding a free segment
- Allocating it
- Returning it when it is no longer needed

RBF uses these routines to manage cluster allocation on disks. They are accessible to users because they are occasionally useful.

See Also

F$DelBit

F$SchBit

F$AllPD — Allocate Process/Path Descriptor

ASM Call

```
OS9 F$AllPD
```

Input

```
(a0) = process/path table pointer
```

Output

```
d0.w = process/path number
(a1) = pointer to process/path descriptor
```

Error Output

```
cc = Carry bit set
d1.w = error code if error
```

State

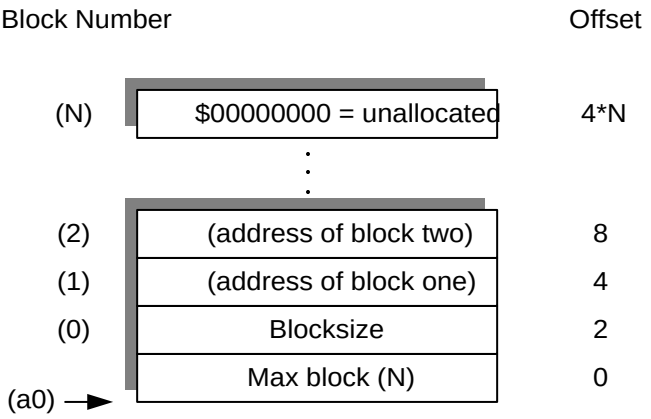
☐ User ☒ System ☐ I/O

Description

F$AllPD allocates fixed-length blocks of system memory. It allocates and initializes (to zeros) a block of storage and returns 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 shown in **Figure D-1**.

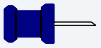
Figure D-1 Map Table Initialization



See Also

F$FindPD

F$RetPD



Note

This is a privileged system-state service request.

F$AllPrc — Allocate Process Descriptor

ASM Call

```
OS9 F$AllPrc
```

Input

None

Output

```
(a2) = Process Descriptor pointer
```

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

F$AllPrc allocates and initializes a process descriptor. The address of the descriptor is kept in the process descriptor table. Initialization consists of:

- Clearing the descriptor
- Setting up the state as system-state
- Marking as unallocated as much of the MMU image as the system allows

On systems without memory management/protection, this is a direct call to F$AllPD.

Possible Errors

E$PrcFul

See Also

F$AllPD

Note

This is a privileged system-state service request.

F$AllTsk — Ensure Protection Hardware Is Ready

ASM Call

```
OS9 F$AllTsk
```

Input

```
(a4) = current process descriptor pointer to allocate
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☐ User ☒ System ☐ I/O

Description

F$AllTsk is called by the kernel just before a process becomes active to insure the protection hardware is ready for the process.

Note

The system-state code making this call should know using the `Trap 0` mechanism (`OS9 F$`) to call this routine causes the current process to be specified, regardless of the value in `(a4)`. If you need to specify a process other than the current one, use the system dispatch tables to make the call directly to the system routine, with the following input parameter:

```
(a4) = process descriptor pointer
```

The System Security Module (SSM) implements F$AllTsk.

See Also

F$DelTsk

F$APROC — Enter Process in Active Process Queue

ASM Call

```
OS9 F$AProc
```

Input

```
(a0) = Address of process descriptor
```

Output

None

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

F$AProc inserts a process into the active process queue so 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 gives up the remainder of its time-slice and the new process runs immediately.

See Also

F$NProc

Process Scheduling in Chapter 2: The Kernel

Note

This is a privileged system-state service request.

F$CCtl — Cache Control

ASM Call

```
OS9 F$CCtl
```

Input

```
d0.l = desired cache control operation
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

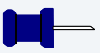
State

☒ User ☐ System ☐ I/O

Description

F$CCtl performs operations on the system instruction and/or data caches, if there are any.

If d0.l is 0, the system instruction and data caches are flushed. Non-super-group, user-state processes may perform this generic operation.



Note

The syscache module implements F$CCtl.

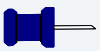
Only system-state processes (for example, device driver) and super-group processes may perform precise operation of F$CCtl. The following bits are defined in d0.l for precise operation:

Table D-6 F$CCtl Bits

Bit	If Set
0	Enable data cache.
1	Disable data cache.
2	Flush data cache.
4	Enable instruction cache.
5	Disable instruction cache.
6	Flush instruction cache.

All other bits are reserved. If any reserved bit is set, an E$Param error is returned.

Any program building or changing executable code in memory should flush the instruction cache by F$CCtl before executing the new code. This is necessary because the hardware instruction cache is not updated by data (write) accesses and may therefore contain the unchanged instruction(s). For example, if a subroutine builds an OS-9 system call on its stack, the F$CCtl system call to flush the instruction cache must execute before executing the temporary instructions.



Note

The cache should be flushed before trying to disable the data cache.

Possible Error

E$Param

F$Chain — Load and Execute New Primary Module

ASM Call

```
OS9 F$Chain
```

Input

```
d0.w = desired module type/language (must be
       program/object or 0=any)
d1.l = additional memory size
d2.l = parameter size
d3.w = number of I/O paths to copy
d4.w = priority
(a0) = module name ptr
(a1) = parameter ptr
```

Output

None. F$Chain does not return to the calling process.

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$Chain executes an entirely new program, but without the overhead of creating a new process. It is similar to a Fork command followed by an Exit. F$Chain effectively resets the calling process' program and data memory areas and begins executing a new primary module. Open paths are not closed or otherwise affected.

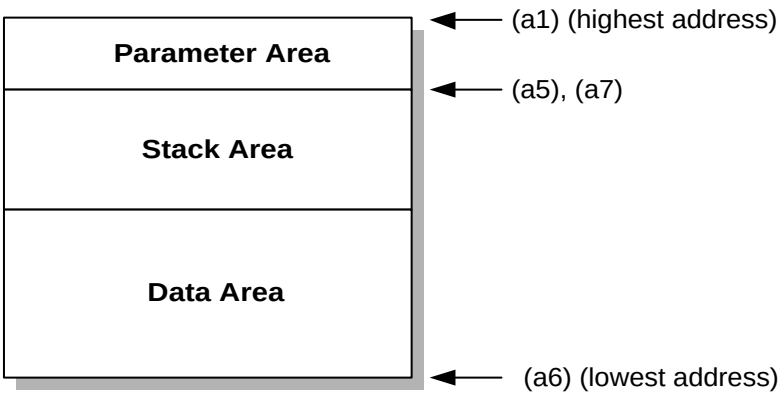
Chain executes as follows:

- Unlink the process' old primary module.

- The system parses the name string of the new process' primary module (the program to be executed). Next, the system module directory is searched to see if a module of 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.
- Reconfigure the data memory area to the specified size in the new primary module's header.
- Erase intercepts and any pending signals.

Figure D-2 shows how `Chain` sets up the data memory area and registers for the new module (these are identical to `F$Fork`).

Figure D-2 Data Memory Area and Registers in Child Process



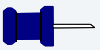
The following registers are passed to the child process:

Table D-7 Registers Passed to the Child Process

Register	Description
sr	n000, where: n = 0 for non-MSP systems n = 1 for MSP systems
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

Table D-7 Registers Passed to the Child Process (continued)

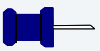
Register	Description
(a5)	Parameter pointer
(a6)	Static storage (data area) base pointer
(a7)	Stack pointer (same as (a5))



Note

(a6) is actually biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. However, it may be significant to note when debugging programs.

The minimum overall data area size is 256 bytes. Address registers point to even addresses.



Note

Most errors occurring during the Chain are returned as an exit status to the parent of the process doing the chain.

Possible Error

E$NEMod

See Also

F$Fork

F$Load

F$ChkMem — Check Access Permissions

ASM Call

```
OS9 F$ChkMem
```

Input

```
d0.l = size of area in bytes
d1.b = permission requested (Read_/Write_/Exec_)
(a2) = address of area to check
(a4) = process descriptor requesting access
```

Error Output

```
cc = carry bit set
d1.w = error code (E$BPAddr) if error (access denied)
```

State

☐ User ☒ System ☐ I/O

Description

F$ChkMem is called by system routines prior to accessing data at the specified address on behalf of a process to determine if the process has access to the specified memory block.

F$ChkMem must check the process' protection image to determine if access to the specified memory area is permitted. F$ChkMem is called by system state routines that may access memory (such as I$Read and I$Write) to verify the user process itself has access to the requested memory. This software check is necessary because the protection hardware is expected to be disabled for system state routines.

Note

If the current system call being checked was made from system-state, you should not make `F$ChkMem` calls, as system-state code is expected to have access to all system memory.

The System Security Module (SSM) implements `F$ChkMem`.

The system-state code making this call should know using the `Trap 0` mechanism (`OS9 F$`) to call this routine causes the current process to be specified, regardless of the value in `(a4)`. If you need to specify a process other than the current one, use the system dispatch tables to make the call directly to the system routine, with the following input parameters:

- `d0.l = size of area in bytes`
- `d1.b = permission requested (Read_/Write_/Exec_)`
- `(a2) = address of area to check`
- `(a4) = process descriptor requesting access`

F$CmpNam — Compare Two Names

ASM 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 = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

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 wildcard characters are recognized in the pattern string:

- Question mark (?) matches any single character
- Asterisk (*) matches any string

The target name must be terminated by a null byte.

Possible Errors

E$Differ	The names do not match.
E$StkOvf	The pattern is too complex.

F$CpyMem — Copy External Memory

ASM 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 = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$CpyMem copies external memory into your buffer for inspection. You can use F$CpyMem to copy portions of the system's address space. This is especially helpful in examining modules. You can view any memory in the system with F$CpyMem.

See Also

F$Move

F$CRC — Generate CRC

ASM 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 = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$CRC generates or checks the CRC (cyclic redundancy check) values of sections of memory. Compilers, assemblers, or other module generators use F$CRC to generate a valid module CRC.

If the CRC of a new module is to be generated, the CRC is accumulated over the entire module, excluding the CRC itself. The accumulated CRC is complemented and then stored in the correct position in the module.

You can calculate the CRC 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 `-1` before the first F$CRC call for any particular module because `accum` is an `int32` value.

An easier method of checking an existing module's CRC is to perform the calculation on the entire module, including the module CRC. The CRC accumulator contains the CRC constant bytes if the module CRC is correct. The CRC constant is defined in `sys.l` and `usr.l` as `CRCCon`. Its value is `$00800fe3`.

Note

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 `0`. For convenience, if a data pointer of `0` is passed, the CRC is updated with one zero data byte. `F$CRC` always returns $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.

See Also

The section on the **The CRC Value** in Chapter 1: System Overview.

F$DatMod — Create Data Module

ASM Call

```
OS9 F$DatMod
```

Input

```
d0.l = size of data required (not including header or
       CRC)
d1.w = desired attr/revision
d2.w = desired access permission
d3.w = desired type/language (optional)
d4.l = memory color type (optional)
(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 = error code if error
```

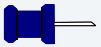
State

☒ User ☐ System ☐ I/O

Description

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 CRC value of 0 and entered into the system module directory. Several processes can communicate with each other using a shared data module.

Be careful not to modify the data module's header or name string to avoid the possibility of the module becoming unknown to the system.



Note

The module created contains at least `d0.l` usable data bytes, but may be somewhat larger. The module itself is larger by at least the size of the module header and CRC, and rounded up to the nearest system memory allocation boundary.

Possible Errors

`E$BNam`

`E$MemFul`

`E$NoRAM`

See Also

`F$Move`

`F$SetCRC`

F$DelBit — De-allocate in Bit Map

ASM 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

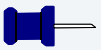
```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$DelBit clears bits in the allocation bit map that were previously allocated and are now free for general use. Bit numbers range from 0 to n-1, where n is the number of bits in the allocation bit map.



Note

The IOMan module implements F$DelBit.

See Also

F$AllBit

F$CpyMem

F$SchBit

F$DExec — Execute Debugged Program

ASM 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 = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$DExec controls the execution of a suspended child process created by the F$DFork call. The process performing F$DExec is suspended and its debugged child process is executed instead. Once the specified

number of instructions are executed, a breakpoint is reached or an unexpected exception occurs, execution terminates and control returns to the parent process. Thus, the parent and the child processes are never active at the same time.

F$DExec traces every instruction of the child process. It checks for the termination conditions after each instruction. Breakpoints are simply lists of addresses to check and work with ROMed object programs. Consequently, the child process being debugged runs at a slow speed.

If a -1 (hex $ffffffff) is passed in d1.l, F$DExec replaces the instruction at each breakpoint address with an illegal opcode. It then executes the child process at full speed (with the trace bit clear) until a breakpoint is reached or the program terminates. This can save an enormous amount of time, but it is impossible for F$DExec to count the number of executed instructions.

Any OS-9 for 68K system calls made by the suspended program are executed at full speed and are considered one logical instruction. The same is true of system-state trap handlers. You cannot debug system-state processes.

The system uses the register buffer passed in the F$DFork call to save and restore the child's registers. Changing the contents of the register buffer alters the child process' registers.

If the child process terminates for any reason, the carry bit is set and returned. Tracing may continue as long as the child process does not perform an F$Exit (even after encountering any normally fatal error). An F$DExit call must be made to return the debugged process' resources (memory).

Note

The Atomic kernel does not support F$DExec.

Note

Tracing is not allowed through user-state trap handlers, intercept routines, and the F$Chain system call. This is not a problem, but may seem strange at times.

Possible Errors

E$IPrcID

E$PrcAbt

See Also

F$DExit

F$DFork

F$DExit — Exit Debugged Program

ASM Call

```
OS9 F$DExit
```

Input

```
d0.w = process ID of child to terminate
```

Output

None

Error Output

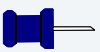
```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$DExit terminates a suspended child process that was created with the F$DFork system call. To permit examination after the process dies, normal termination by the child process does not release any of its resources.



Note

The Atomic kernel does not support F$DExit.

Possible Errors

E$IPrcID

See Also

F$DExec
F$DFork
F$Exit

F$DFork — Fork Process Under Control of Debugger

ASM Call

```
OS9 F$DFork
```

Input

```
d0.w = desired module type/revision (0 = any)
d1.l = additional stack space to allocate (if any)
d2.l = parameter size
d3.w = number of I/O paths for child to inherit
d4.w = module priority
(a0) = module name ptr (or pathlist)
(a1) = parameter ptr
(a2) = register buffer: copy of child's
       (d0-d7/a0-a7/sr/pc)
```

Output

```
d0.w = child process ID
(a0) = updated past module name string
(a2) = initial image of the child process's registers
       in buffer
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

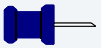
Description

F$DFork is similar to F$Fork, except F$DFork creates a process whose execution can be closely controlled. The child process is not placed in the active queue but is left in a suspended state. This allows

the debugger to control its execution through the special system calls F$DExec and F$DExit. (The child process is created with the trace bit of its status register set and is executed with the F$DExec system call.)

The register buffer is an area in the caller's data area permanently associated with each child process. It is set to an image of the child's initial registers for use with the F$DExec call.

For information about process creation, see the F$Fork service request.



Note

A process created by F$DFork does not execute unless it is told to do so. When a process is run, the trace bit is set in the user status register. This causes the system trace exception handler to occur once for each user instruction executed, thus user programs run slowly.

Processes whose primary module is owned by a super-user may only be debugged by a super-user. You cannot debug system-state processes.

Note

The Atomic kernel does not support F$DFork.

See Also

F$DExec

F$DExit

F$Fork

F$DelPrc

De-allocate Process Descriptor Service Request

ASM Call

```
OS9 F$DelPrc
```

Input

```
d0.w = process ID to de-allocate
```

Output

None

Error Output

```
cc = carry set
d1.w = appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

F$DelPrc de-allocates a process descriptor previously allocated by F$AllPD. You must ensure any system resources used by the process are returned before calling F$DelPrc.

Currently, the F$DelPrc request is simply a convenient interface to the F$RetPD service request. It is preferred to F$RetPD to ensure compatibility with future releases of the operating system needing to perform process specific de-allocations.

Possible Errors

E$BNam

E$KwnMod

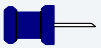
See Also

F$AllPD

F$AllPrc

F$FindPD

F$RetPD



Note

This is a privileged system-state service request.

F$DelTsk

ASM Call

```
OS9 F$DelTsk
```

Input

```
(a4) = process descriptor pointer to release
```

Error Output

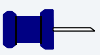
```
cc = carry bit set
d1.w = error code if error
```

State

☐ User ☒ System ☐ I/O

Description

F$DelTsk is called by the kernel when a process terminates to return the process' protection resources.



Note

The system-state code making this call should know using the `Trap 0` mechanism (`OS9 F$`) to call this routine causes the current process to be specified, regardless of the value in `(a4)`. If you need to specify a process other than the current one, use the system dispatch tables to make the call directly to the system routine, with the following input parameter:

```
(a4) = process descriptor pointer
```

The System Security Module (SSM) implements F$DelTsk.

Note

The OS-9 kernel ignores all errors returned by `F$DelTsk`.

See Also

`F$AllTsk`

Note

This is a privileged system-state service request.

F$Event — Create, Manipulate, and Delete Events

ASM Call

```
OS9 F$Event
```

Input

```
d1.w = Event function code
All others are dependent on function code
```

Output

Dependent on function code

Error Output

Dependent on function code

State

☒ User ☐ System ☐ I/O

Description

Events are multiple-value semaphores that synchronize concurrent processes sharing resources such as files, data modules, and CPU time. F$Event provides facilities:

- To create and delete events.
- To permit processes to link/unlink events and obtain event information.
- To suspend operation until an event occurs.
- For various means of signaling.

An OS-9 event is a 32-byte system global variable maintained by the system. The following fields are included in each event:

Table D-8 Event Fields

Event	Description
`Event ID`	This number and the event's array position are used to create a unique ID.
`Event name`	This name must be unique and cannot exceed 12 characters.
`Event value`	This four-byte integer value has a range of two billion.
`Wait increment`	This value is added to the event value when a process waits for the event. It is set when the event is created and does not change.
`Signal increment`	This value is added to the event value when the event is signaled. This value is set when the event is created and does not change.
`Link Count`	This is the event use count.
`Next event`	This is a pointer to the next process in the event queue. An event queue is circular and includes all processes waiting for the event. Each time the event is signaled, this queue is searched.
`Previous event`	This is a pointer to the previous process in the event queue.

The following function codes are supported:

Table 7-5 Event Function Codes

Name	Description
Ev$Creat	Create new event.
Ev$Delet	Delete existing event.
Ev$Info	Return event information.
Ev$Link	Link to existing event by name.
Ev$Pulse	Signal an event occurrence.
Ev$Read	Read event value without waiting.
Ev$Set	Set event variable and signal an event occurrence.
Ev$SetR	Set relative event variable; signal an event occurrence.
Ev$Signl	Signal an event occurrence.
Ev$UnLnk	Unlink event.
Ev$Wait	Wait for event to occur.
Ev$WaitR	Wait for relative to occur.

Possible Errors

Dependent on function code

See Also

The section on events in Chapter 1: System Overview.

Ev$Creat — Create New Event

Input

```
d0.l = initial event variable value
d1.w = 2 (Ev$Creat function code)
d2.w = auto-increment for Ev$Wait
d3.w = auto-increment for Ev$Signl
(a0) = event name string pointer (max 11-chars)
```

Output

```
d0.l = event ID number
(a0) = updated past event name
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

You can create and delete events dynamically as needed. When an event is created, an initial signed value is specified, as well as signed increments to be applied each time the event occurs or is waited for. The returned event ID number is used in subsequent F$Event calls to refer to the event created.

Possible Errors

E$BNam	Name is syntactically incorrect or longer than 11 chars.
E$EvFull	The event table is full.
E$EvBusy	The named event already exists.

Ev$Delet — Delete Existing Event

Input

```
(a0) = event name string pointer (max 11-chars)
d1.w = 3 (Ev$Delet function code)
```

Output

```
(a0) = updated past event name
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$Delet` removes an event from the system event table, freeing the entry for use by another event. Events have an implicit use count (initially set to 1), that is incremented with each `Ev$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.

Possible Errors

`E$BNam`	Name is syntactically incorrect or longer than 11 chars.
`E$EvNF`	Event not found in the event table.

E$EvBusy	The event has a non-zero link count.

Ev$Info — Return Event Information

Input

```
d0.l = event index (ID number) to begin search
d1.w = 7 (Ev$Info function code)
(a0) = ptr to buffer for event information
```

Output

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

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

Ev$Info returns a copy of the 32-byte event table entry associated with an event. Unlike other F$Event functions, Ev$Info only uses the low word of d0. This index is the system event number, ranging from zero to the maximum number of system events minus one. The event information block for the first active event with an index greater than or equal to this index is returned in the caller's buffer. If none exists, an error is returned. Ev$Info is provided for utilities needing to determine the status of all active events.

Possible Errors

E$EvntID	The index is above all active events.

Ev$Link — Link to Existing Event by Name

Input

```
(a0) = event name string pointer (max 11 chars)
d1.w = 0 (Ev$Link function code)
```

Output

```
d0.l = event ID number
(a0) = updated past event name
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

Ev$Link determines the ID number of an existing event. Once an event is linked, all subsequent references are made using the event ID returned. This permits the system to access events quickly, while protecting against programs using invalid or deleted events. The event use count is incremented when an Ev$Link is performed. To keep the use count synchronized properly, perform an Ev$UnLnk when the event will no longer be used.

Possible Errors

E$BNam	Name is syntactically incorrect or longer than 11 chars.
E$EvNF	Event not found in the event table.

Ev$Pulse — Signal Event Occurrence

Input

```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = 9 (Ev$Pulse function code)
d2.l = event pulse value
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$Pulse` signals an event occurrence, but differs from `Ev$Signl`. The event variable is set to the value passed in `d2`, and the signal auto-increment is not applied. Then, the `Ev$Signl` search routine is executed and the original event value is restored.

Possible Errors

`E$EvntID`	The ID specified is not a valid active event.

Ev$Read — Read Event Value Without Waiting

Input

```
d0.l = event ID number
d1.w = 6 (Ev$Read function code)
```

Output

```
d1.l = current event value
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

Ev$Read reads the value of an event without waiting or modifying the event variable. You can use this to determine the availability of the event (or associated resource) without waiting.

Possible Errors

E$EvntID	ID specified is not a valid active event.

Ev$Set

Set Event Variable and Signal Event Occurrence

Input

```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = A (Ev$Set function code)
d2.l = new event value
```

Output

```
d1.l = previous event value
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

Ev$Set is similar to the Ev$Signl call, except the event variable is initially set to the value passed in d2 rather than updated with the signal auto-increment. After this is done, the Ev$Signl routine is executed directly.

Possible Errors

E$EvntID	The ID specified is not a valid active event.

Ev$SetR — Set Relative Event Variable and Signal Event Occurrence

Input

```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = B (Ev$SetR function code)
d2.l = (signed) increment for event variable
```

Output

```
d1.l = previous event value
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

Ev$SetR is similar to Ev$Signl, but instead of using the signal auto-increment value to update the event variable, the value in d2 is used. Arithmetic underflows or overflows are set to $80000000 or $7fffffff, respectively.

Possible Errors

E$EvntID	The ID specified is not a valid active event.

Ev$Signl — Signal Event Occurrence

Input

```
d0.l = event ID number
d1.w = MS bit set to activate all processes in range
       LS bits = 8 (Ev$Signl function code)
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$Signl` signals an event has occurred. The current event variable is 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` (the function code) is set, all processes in the event queue with a value in range are activated. The sequence is the same for each event in the queue until the queue is exhausted:

- The signal auto-increment is added to the event variable
- The first process in range is awakened
- The event variable is updated with the wait auto-increment
- The search continues with the updated value

Possible Errors

`E$EvntID`	The ID specified is not a valid active event.

Ev$UnLnk — Unlink Event

Input

```
d0.l = event ID number
d1.w = 1 (Ev$UnLnk function code)
```

Output

None

Error Output

```
cc = carry bit se
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$Unlnk` informs the system a process no longer uses an event. The event use count is decremented. When it reaches zero, you must delete it. OS-9 uses this only for error checking.

Possible Errors

`E$EvntID`	ID specified is not a valid active event.

Ev$Wait — Wait for Event to Occur

Input

```
d0.l = event ID number
d1.w = 4 (Ev$Wait function code)
d2.l = minimum activation value (signed)
d3.l = maximum activation value (signed)
```

Output

```
d1.l = actual event value
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$Wait` waits for an event to occur. The event variable is compared to the range specified in `d2` and `d3`. If the value is not in range, the calling process is suspended in a FIFO event queue. It waits until an `Ev$Signl` occurs putting the value in range and adding the wait auto-increment (specified at creation) 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 is not within the specified range. The caller's intercept routine is executed, but an event error is not returned.

Possible Errors

`E$EvntID`	ID specified is not a valid active event.

Ev$WaitR — Wait for Relative Event to Occur

Input

```
d0.l = event ID number
d1.w = 5 (Ev$WaitR function code)
d2.l = minimum relative activation value (signed)
d3.l = maximum relative activation value (signed)
```

Output

```
d1.l = actual event value
d2.l = minimum actual activation value
d3.l = maximum actual activation value
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

`Ev$WaitR` works exactly like `Ev$Wait`, except 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.

Possible Errors

`E$EvntID`	ID specified is not a valid active event.

F$Exit — Terminate Calling Process

ASM Call

```
OS9 F$Exit
```

Input

```
d1.w = Status code to be returned to parent process
```

Output

Process is terminated

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$Exit is the means by which a process can terminate itself. Its data memory area is de-allocated and its primary module is unlinked. All open paths are automatically closed.

The parent can execute an F$Wait call to detect the death of the process. This returns (to the parent) the status word passed by the child in its exit call. The shell assumes the status word is an OS-9 error code that the terminating process wishes to pass back to its parent process. The status word could also be a user-defined status value.

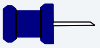
Processes called directly by the shell should only return an OS-9 error code or zero if no error occurred.

Note

The parent ***must*** do an F$Wait before the process descriptor is returned.

An F$Exit call functions as follows:

- Close all paths
- Return memory to the system
- Unlink the primary module and user trap handlers
- Free process descriptor of any dead child processes
- If the parent is dead, free the process descriptor
- If the parent has not executed an F$Wait call, leave the process in limbo until the parent notices the death
- If the parent is waiting, move the parent to the active queue, inform the parent of the death/status, remove the child from the sibling list, and free its process descriptor memory



Note

Only the primary module and the user trap handlers are unlinked. Unlink any other modules that are loaded or linked by the process before calling F$Exit.

Although F$Exit closes any open paths, it pays no attention to errors returned by the I$Close request. Because of I/O buffering, this can cause write errors to go unnoticed when paths are left open. However, by convention, the standard I/O paths (0, 1, 2) are usually left open.

See Also

F$AProc
F$FindPD
F$Fork
F$RetPD
F$SRtMem
F$UnLink
F$Wait
I$Close

F$FindPD — Find Process/Path Descriptor

ASM Call

```
OS9 F$FindPD
```

Input

```
d0.w = process/path number
(a0) = process/path table pointer
```

Output

```
(a1) = pointer to process/path descriptor
```

Error Output

```
cc = Carry bit set
d1.w = error code if error
```

State

☐ User ☒ System ☐ I/O

Description

F$FindPD converts a process or path number to the absolute address of its descriptor data structure. You can use it for simple memory management of fixed length blocks. See F$AllPD for a description of the data structure used.

See Also

F$AllPD

F$RetPD

Note

This is a privileged system-state service request.

F$FIRQ — Add or Remove Device from Fast IRQ Table (System-State Only)

ASM Call

```
OS9 F$FIRQ
```

Input

```
d0.b = vector number
      (25 - 31 for autovectors,
       57 - 63 for 68070 on-chip autovectors,
       64 - 255 for vectored IRQs.)
d1.b = reserved, must be 0.
(a0) = IRQ service routine entry point (0 = remove)
(a2) = device static storage.
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code
```

State

☐ User ☒ System ☐ I/O

Description

F$FIRQ installs an IRQ service routine into or removes one from the system's fast IRQ table. This fast IRQ system provides a faster interrupt response context than the normal IRQ vector polling scheme (provided via F$IRQ).

To:

- place a routine into the fast IRQ system, set a0 to a non-zero value.
- remove a routine from the fast IRQ system, set a0 to zero.

Only one F$FIRQ routine can be active at a time per vector. An attempt to install a second routine on a vector using F$FIRQ causes an E$VctBsy error. If additional devices are required to be on the same vector as an F$FIRQ device, use F$IRQ to install them.

Device drivers are required to determine if their device caused the interrupt. Service routines conform to the following register conventions:

Input

```
d0.w = vector address (vector number * 4)
(a2) = device static storage pointer
(a6) = system global data pointer (D_'s)
(a7) = system stack (small IRQ default stack)
```

Output

```
cc = carry clear: return to interrupted context
cc = carry set: poll further devices on vector using
                F$IRQ polling system.
```

WARNING

Fast interrupt service routines may destroy the d0 and a2 registers. You must preserve all other registers used.

The interrupt stack provided is a small ***default*** IRQ stack. The service routine must ensure this stack is not overflowed. The Init module's ***IRQ stack*** value does ***not*** affect this stack; the Init module value is used to control the F$IRQ polling system stack.

Possible Errors

E$VctBsy

See Also

F$IRQ

The ***OS-9 for 68K Processors Technical I/O Manual*** contains more information on RBF and SCF device drivers.

F$Fork — Create New Process

ASM 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 = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$Fork creates a new process that 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 initially 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 to be loaded into memory. The first module in this file is linked and executed. To be loaded, the module must be program object code and have the appropriate read and/or execute permissions set for the user.

The primary module's module header is used to determine the process' initial data area size. OS-9 then attempts to allocate RAM equal to the required data storage size plus any additional size specified in `d1`, plus the size of any parameter passed. The RAM area must be contiguous.

The new process' registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module's entry point. If `d4.w` is set to zero, the new process inherits the same priority as the calling process.

When the shell processes a command line, it passes a copy of the parameter portion (if any) of the command line as a parameter string. The shell appends an end-of-line character to the parameter string to simplify string-oriented processing.

If any of the these operations are unsuccessful, the fork is aborted and an error is returned to the caller. The following diagram shows how `F$Fork` sets up the data memory area and registers for a newly-created process. For more information, see `F$Wait`.

Figure D-3 Data Memory Area and Registers in Child Process

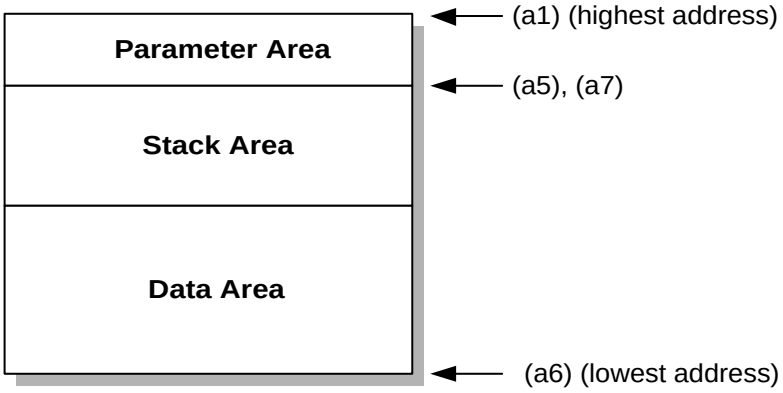


Table D-9 Registers Passed to the Child Process

Register	Description
sr	n000, where: n = 0 for non-MSP systems n = 1 for MSP systems
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

Table D-9 Registers Passed to the Child Process (continued)

Register	Description
(a5)	Parameter pointer
(a6)	Static storage (data area) base pointer
(a7)	Stack pointer (same as (a5))

Note

(a6) is actually biased by $8000, but this can usually be ignored because the linker biases all data references by -$8000. However, it may be significant to note when debugging programs.

Note

Both the child and parent process execute concurrently. If the parent executes an F$Wait call immediately after the fork, it waits until the child dies before it resumes execution. A child process descriptor is returned only when the parent does an F$Wait call.

Modules owned by a super-user execute in system state if the system-state bit in the module's attributes is set. This is rarely necessary, quite dangerous, and not recommended for beginners.

Possible Errors

E$IPrcID

See Also

F$Chain
F$Exit
F$Wait

F$GBlkMp — Get Free Memory Block Map

ASM Call

```
OS9 F$GblkMp
```

Input

```
d0.l = Address to begin reporting segments
d1.l = Size of buffer in bytes
(a0) = Buffer pointer
```

Output

```
d0.l = System’s minimum memory allocation size
d1.l = Number of memory fragments in system
d2.l = Total RAM found by system at startup
d3.l = Current total free RAM available
(a0) = Memory fragment information
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

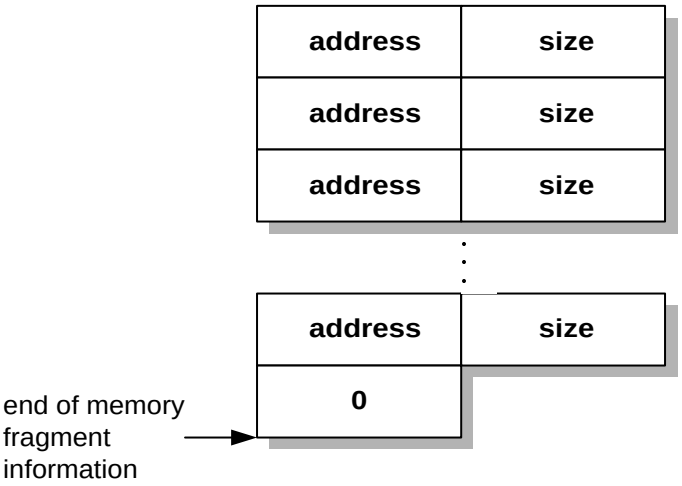
State

☒ User ☐ System ☐ I/O

Description

F$GBlkMp copies the address and size of the system's free RAM blocks into the user's buffer for inspection. It also returns various information concerning the free RAM as noted by the output registers above. The address and size of the free RAM blocks are returned in the user's buffer in following format (address and size are 4-bytes):

Figure D-4



Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be accessed directly. Use F$SRqMem to request free memory blocks.

Note

F$GBlkMp provides a status report concerning free system memory for mfree and similar utilities. The address and size of free RAM changes with system use. Although F$GblkMp returns the address and size of the system's free memory blocks, these blocks may never be accessed directly. Use F$SRqMem to request free memory blocks.

See Also

F$Mem

F$SRqMem

F$GModDr — Get Copy of Module Directory

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

`F$GModDr` copies the system's module directory into the user's buffer for inspection. `mdir` uses `F$GModDr` to look at the module directory. Although the module directory contains pointers to each module in the system, you should never directly access the modules. Rather, use `F$CpyMem` 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.

Note

F$GModDr is provided primarily for use by mdir and similar utilities. The format and contents of the module directory may change on different releases of OS-9 for 68K. For this reason, it is often preferable to use the output of mdir to determine the names of modules in memory.

See Also

F$CpyMem

F$Move

F$GPrDBT — Get Copy of Process Descriptor Block Table

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$GPrDBT copies the process descriptor block table into the caller's buffer for inspection. The `procs` utility uses F$GPrDBT to quickly determine which processes are active in the system. Although F$GPrDBT returns pointers to the process descriptors of all processes, never access the process descriptors directly. Instead, use the F$GPrDsc system call if you need to inspect particular process descriptors.

The system call, F$AllPD, describes the format of the process descriptor block table.

See Also

F$AllPD

F$GPrDsc

F$GPrDsc — Get Copy of Process Descriptor

ASM 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

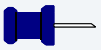
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

`F$GPrDsc` copies a process descriptor into the caller's buffer for inspection. There is no way to change data in a process descriptor. The `procs` utility uses `F$GPrDsc` to gain information about an existing process.



Note

The format and contents of a process descriptor may change with different releases of OS-9 for 68K.

Possible Errors

E$IPrcID

See Also

F$GPrDBT

F$GProcP — Get Process Pointer

ASM Call

```
OS9 F$GProcP
```

Input

```
d0.w = requested process ID
```

Output

```
(a1) = process descriptor pointer
```

Error Output

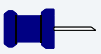
```
cc = Carry bit set
d1.w = appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

F$GProcP returns a pointer to the process descriptor associated with the requested process ID. It is used by system-state code to inspect and possibly change the contents of a process descriptor.



Note

The format and contents of a process descriptor may change with different releases of OS-9 for 68K.

If you use this call, be aware of having the process die while using this pointer. To prevent the process descriptor from being deleted, ensure system-state preemption for your process is disabled while using the pointer. To do this, set the P$Preempt field of your process descriptor.

Possible Errors

E$IPrcID

Note

This is a privileged system-state service request.

F$Gregor — Get Gregorian Date

ASM Call

```
OS9 F$Gregor
```

Input

```
d0.l = time (seconds since midnight)
d1.l = Julian date
```

Output

```
d0.l = time (00hhmmss)
d1.l = date (yyyymmdd)
```

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

F$Gregor converts Julian dates to Gregorian dates. Gregorian dates are considered the normal calendar dates.

The Julian date is similar to the Julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

Note

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 for 68K makes this adjustment on October 15, 1582. Be careful when you are working with old dates, because the same day may be recorded as a different date by different sources.

Note

`F$Gregor` is the inverse function of `F$Julian`.

See Also

`F$Julian`

`F$Time`

F$GSPUMp — Return Status of Access Permissions

ASM Call

```
OS9 F$GSPUMp
```

Input

```
d0.w = process ID for which to get information
d2.l = size of area for information
(a0) = address of area for information
```

Output

```
d2.l = size of area for information
(a0) = address of area for information
```

Error Output

```
cc = carry bit set
d1.w = error code if error (access denied)
```

State

☒ User ☐ System ☐ I/O

Description

F$GSPUMp returns data about a specified process' memory map for debugging purposes.

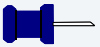
The format of the data is:

8 bits: `00000ewr`(high order)

8 bits: `use count`(low order)

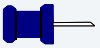
One word in this format is returned for each memory block in the system. This information is taken from the process' SSM data structure. If the address space given is not big enough, only the information that fits in the area is returned.

F$GSPUMp is used primarily by the maps utility to display process image information.



Note

The System Security Module (SSM) implements F$GSPUMp.



Note

The buffer pointer must be word-aligned to avoid an E$AdrErr.

Possible Errors

E$AdrErr

E$IPrcID

E$Param

E$UnkSvc (non SSM systems).

F$Icpt — Set Up Signal Intercept Trap

ASM Call

```
OS9 F$Icpt
```

Input

```
(a0) = Address of the intercept routine
(a6) = Address to be passed to the intercept routine
```

Output

```
(d0) = Number of signals pending
```

Error Output

None

State

☒ User ☐ System ☐ I/O

Description

`F$Icpt` tells OS-9 to install a signal intercept routine:

- `(a0)` contains the address of the signal handler routine.
- `(a6)` usually contains the address of the program's data area.

Note

Signals sent to the process causes the intercept routine to be called instead of the process being killed.

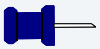
After the `F$Icpt` call has been made, whenever the process receives a signal, its intercept routine executes. A signal aborts a process that has not used the `F$Icpt` service request and its termination status

(register d1.w) is 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's data area. Avoid complicated system calls (such as I/O). After the intercept routine is complete, it may return to normal process execution by executing the F$RTE system call.



Note

Each time the intercept routine is called, 70 bytes are used on the user's stack.

See Also

F$RTE

F$Send

F$SigReset

F$ID — Get Process ID / User ID

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$ID returns the caller's process ID number, group and user ID, and current process priority (all word values). OS-9 assigns a unique process ID 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$IODel — Check For Busy I/O Module

ASM Call

```
OS9 F$IODel
```

Input

```
(a0) = module pointer
```

Output

None

Error Output

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

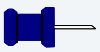
State

☐ User ☒ System ☐ I/O

Description

F$IODel is called by the kernel when an I/O module is unlinked for the last time. F$IODel checks the system's device table to see if the module is busy.

If the module is not busy, no error is returned and the caller may remove the module. If the module is busy, an error is returned and the caller needs to keep the link count for the module at 1.



Note

The IOMan module implements F$IODel.

Possible Errors

E$ModBsy

Note

This is a privileged system-state service request.

F$IOQu — Enter I/O Queue

ASM Call

```
OS9 F$IOQu
```

Input

```
d0.w = Process Number
```

Output

None

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

F$IOQu links the calling process into the I/O queue of the specified process and performs an untimed sleep. It is assumed routines associated with the specified process send a wake up 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 performs an F$IOQu call to the process in control of the device. When the first process returns from the file manager, the kernel automatically wakes up the IOQu-ed process.

Note

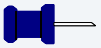
The IOMan module implements F$IOQu.

See Also

F$FindPD

F$Send

F$Sleep



Note

This is a privileged system-state service request.

F$IRQ — Add or Remove Device from IRQ Table

ASM Call

```
OS9 F$IRQ
```

Input

```
d0.b = vector number
       25-31 for autovectors
       57-63 for 68070 on-chip autovectors
       64-255 for vectored IRQs
d1.b = priority (0 = polled first, 255 = last)
(a0) = IRQ service routine entry point (0 = delete)
(a2) = device static storage
(a3) = port address
```

Output

None

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

F$IRQ installs an IRQ service routine into the system polling table. If (a0) equals 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 0, only this device is allowed to use the vector. Otherwise, any vector may support multiple devices. OS-9 does not poll the I/O port before 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:

```
d0.w = vector offset (vector number *4)
(a2) = global static pointer
(a3) = port address
(a6) = system global data pointer (D_'s)
(a7) = system stack (in active proc's descriptor)
```

Output:

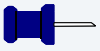
None

Error Output

Carry bit set if the device did not cause the interrupt.

WARNING

Interrupt service routines may destroy the following registers: d0, d1, a0, a2, a3, and/or a6. You must preserve all other registers used.



Note

Note the following:

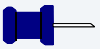
- You may not put zero priority multiple auto-vectored devices on the polling list.
- Zero priority devices do no exclude use of the vector by F$FIRQ devices.

Possible Errors

E$POLL is returned if the polling table is full.

See Also

The ***OS-9 for 68K Processors Technical I/O Manual*** contains more information on RBF and SCF device drivers.



Note

This is a privileged system-state service request.

F$Julian — Get Julian Date

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

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, subtract the lower Julian date number from the higher number.

The Julian day number returned is similar to the Julian date used by astronomers. It is based on the number of days that have elapsed since January 1, 4713 B.C. Each astronomical Julian day changes at noon. OS-9 differs slightly from the astronomical standard by changing Julian dates at midnight. It is relatively easy to adjust for this, when necessary.

You can also use the Julian day number to determine the day of the week for a given date. Use the following formula:

```
weekday = MOD(Julian_Date + 2, 7)
```

This returns the day of the week as 0 = Sunday, 1 = Monday, etc.

Note

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. Be careful when working with old dates, because the same day may be recorded as a different date by different sources.

F$Link — Link to Memory Module

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

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, you can obtain this and other information from the module header. The module's link count is incremented to keep track of how many processes are using the module. If the module requested is not re-entrant, only one process may link to it at a time.

If the module's access word does not give the process read permission, the link call fails. Link also fails to find modules whose header has been destroyed (altered or corrupted) in memory.

Possible Errors

E$BNam

E$MNF

E$ModBsy

See Also

F$Load

F$UnLink

F$UnLoad

F$Load — Load Module(s) from File

ASM Call

```
OS9 F$Load
```

Input

```
d0.b = Access mode
d1.l = Memory "color" type to load (optional)
(a0) = Pathname string 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
```

State

☒ User ☐ System ☐ I/O

Description

F$Load opens a file specified by the pathlist. It reads one or more memory modules from the file into memory until it reaches an error or end of file. Then, it closes the file. Modules are usually loaded into the highest physical memory available.

Note

The IOMan module implements `F$Load`.

An error can indicate the following:

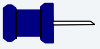
- An actual I/O error.
- A module with a bad parity or CRC.
- The system memory is full.

All loaded modules are added to the system module directory, and the first module read is linked. The parameters returned are the same as those returned by a link call, and apply only to the first module loaded.

To be loaded, the file must contain a module or modules possessing a proper module header and CRC. The access mode may be specified as either `Exec_` or `Read_`, causing the file to load 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 prevents normal users from executing privileged service requests.

The input register specifying memory color type (`d1.l`) is only referenced if the most significant bit of `d0.b` is set.



Note

`F$Load` does not work on SCF devices.

Possible Errors

`E$BMID`

`E$MemFul`

F$Mem — Resize Data Memory Area

ASM 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

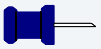
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

F$Mem contracts the process' data memory area. The new size requested is rounded up to an even memory allocation block (16 bytes minimum). Additional memory is allocated contiguously upward (towards higher addresses), or de-allocated downward from the old highest address. If d0 equals 0, the call is considered an information request and the current upper bound and size is returned.

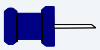


Note

If you wish to expand memory, you should use F$SRqMem.

This request can never return all of a process' memory, or cause de-allocation of memory at its current stack pointer.

The request may return an error upon an expansion request even though adequate free memory exists, because the data area must always be contiguous. Memory requests by other processes may fragment memory into smaller, scattered blocks that are not adjacent to the caller's present data area.



Note

`F$Mem` calls to resize the data area always fail for versions of the kernel from OS-9 for 68K V2.3 and greater. Only an information request (`d0=0`) works on OS-9 for 68K V2.3 and greater.

Possible Errors

`E$DelSP`

`E$MemFul`

`E$NoRAM`

F$Move — Move Data (Low Bound First)

ASM Call

```
OS9 F$Move
```

Input

```
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
```

State

☐ User ☒ System ☐ I/O

Description

F$Move is a fast ***block-move*** subroutine capable of copying data bytes from one address space to another (usually from system to user or vice versa).

The data movement subroutine is optimized to make use of long moves whenever possible. If the source and destination buffers overlap, an appropriate move (left to right or right to left) is used to avoid loss of data due to incorrect propagation.

Note

This is a privileged system-state service request.

F$NProc — Start Next Process

ASM Call

```
OS9 F$NProc
```

Input

None

Output

Control does not return to caller.

Error Output

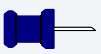
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☐ User ☒ System ☐ I/O

Description

F$NProc 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.



Note

The process calling NProc should already be in one of the system's process queues. If it is not, the calling process becomes unknown to the system even though the process descriptor still exists and is printed out by a procs command.

See Also

F$AProc

Note

This is a privileged system-state service request.

F$Panic — System Catastrophic Occurrence

ASM Call

```
OS9 F$Panic
```

Input

```
d0.l = panic code
```

Output

None. F$Panic generally does not return.

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

The OS-9 kernel makes a F$Panic request when it detects a disastrous, but not necessarily fatal, system condition. Ordinarily, F$Panic is undefined and the system dies.

The system administrator may install a service routine for F$Panic as part of an OS9P2 startup module. The function of such a routine might be to fork a warmstart Sysgo process or to cause the system to re-boot.

Two panic codes are defined:

Table D-10 Panic Codes

Code	Description
K$Idle	The system has no processes to execute.
K$PFail	Power failure has been detected.

F$Panic is called only when the kernel believes there are no processes remaining to be executed. Although it is likely the system is dead at this point, it may not be. Interrupt service routines or system-state alarms could cause the system to become active.

Note

The OS-9 kernel does not detect power failure. However, some machines are equipped with hardware that can detect power failure. For these machines, you could install an OS9P2 routine to call F$Panic when power failure occurs.

See Also

F$SSvc

The section on installing system-state routines in Chapter 2: The Kernel.

F$Permit (System-State) Allow Process Access to Specified Memory

ASM Call

```
OS9 F$Permit
```

Input

```
d0.l = size of area in bytes
d1.b = permission requested (Read_/Write_/Exec_)
(a2) = address of area requested
(a3) = SSM global static storage
(a4) = process descriptor removing access
(a6) = system global pointer
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☐ User ☒ System ☐ I/O

Description

F$Permit is called primarily by the kernel to permit a process access to its data and code space on SSM systems. It is also invoked when a process requests additional memory (F$SRqMem) or links to another module so memory can be accessed by the process.

Note

The system-state code making this call should know using the `Trap 0` mechanism (`OS9 F$`) to call this routine causes the current process to be specified, regardless of the value in `(a4)`. If you need to specify a process other than the current one, use the system dispatch tables to make the call directly to the system routine, with the following input parameters:

- `d0.l = size of area in bytes`
- `d1.b = permission requested (Read_/Write_/Exec_)`
- `(a2) = address of area requested`
- `(a3) = SSM global static storage`
- `(a4) = process descriptor removing access`
- `(a6) = system global pointer`

The System Security Module (SSM) implements `F$Permit`.

See Also

`F$Protect (System-State)`

F$Permit (User-State) Allow Process Access to Specified Memory

ASM Call

```
OS9 F$Permit
```

Input

```
d0.l = size of area in bytes
d1.b = permission requested (Read_/Write_/Exec_)
(a2) = address of area requested
(a4) = process descriptor allowing access
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

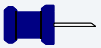
State

☒ User ☐ System ☐ I/O

Description

F$Permit permits a process to access to a specified memory region. It is used primarily on memory protected (SSM) systems. Systems with SSM installed only allow a process to access its own data and code spaces. Processes requiring access to memory regions outside of their code/data area (for example, memory mapped video) should use F$Permit before any attempt to access the region.

Only super-group users (0.n) may call F$Permit from user-state.



Note

The System Security Module (SSM) implements F$Permit.

Possible Errors

E$Permit

See Also

F$Protect (User-State)

F$PErr — Print Error Message

ASM Call

```
OS9 F$PErr
```

Input

```
d0.w = Error message path number (0=none)
d1.w = Error number
```

Output

None

Error Output

None

State

☒ User ☐ System ☐ I/O

Description

F$PErr is the system's error reporting facility. It writes an error message to the standard error path. Most OS-9 systems print ERROR #mmm.nnn. Error numbers 000:000 to 063:255 are reserved for the operating system.

If an error path number is specified, the path is searched for a text description of the error encountered. The error message path contains an ASCII file of error messages. Each line may be up to 80 characters long. If the error number matches the first seven characters in a line (000:215), the rest of the line is 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 to delete a currently open file. Anytime a file already in use is opened for non-sharable access, this error occurs. It also occurs if you try to access a non-sharable device (for example, a printer) that is busy.

Note

The IOMan module implements `F$PErr`.

F$Protect (System-State) Remove Process' Permission to Memory Block

ASM Call

```
OS9 F$Protect
```

Input

```
d0.l = size of area in bytes
(a2) = address of area returned
(a3) = SSM global static storage
(a4) = process descriptor removing access
(a6) = system global pointer
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☐ User ☒ System ☐ I/O

Description

F$Protect is primarily called by the kernel to remove access permission to a block of memory. This occurs when the process exits or when it unlinks a module or returns extra memory resources (F$SRtMem).

Note

The system-state code making this call should know using the `Trap 0` mechanism (`OS9 F$`) to call this routine causes the current process to be specified, regardless of the value in `(a4)`. If you need to specify a process other than the current one, use the system dispatch tables to make the call directly to the system routine, with the following input parameters:

- `d0.l = size of area in bytes`
- `(a2) = address of area requested`
- `(a3) = SSM global static storage`
- `(a4) = process descriptor removing access`
- `(a6) = system global pointer`

The System Security Module (SSM) implements `F$Protect`

Possible Errors

`E$Permit`

See Also

`F$Permit (System-State)`

F$Protect (User-State) — Remove Process' Permission to Memory Block

ASM Call

```
OS9 F$Protect
```

Input

```
d0.l = size of area in bytes
(a2) = address of area returned
```

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$Protect removes process access permission to a specified region (usually permitted via F$Permit (User-State)).

Only super-group users (0.n) may call F$Protect from user-state.

Note

The System Security Module (SSM) implements F$Protect.

Possible Errors

E$Permit

See Also

F$Permit (User-State)

F$PrsNam — Parse Path Name

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

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

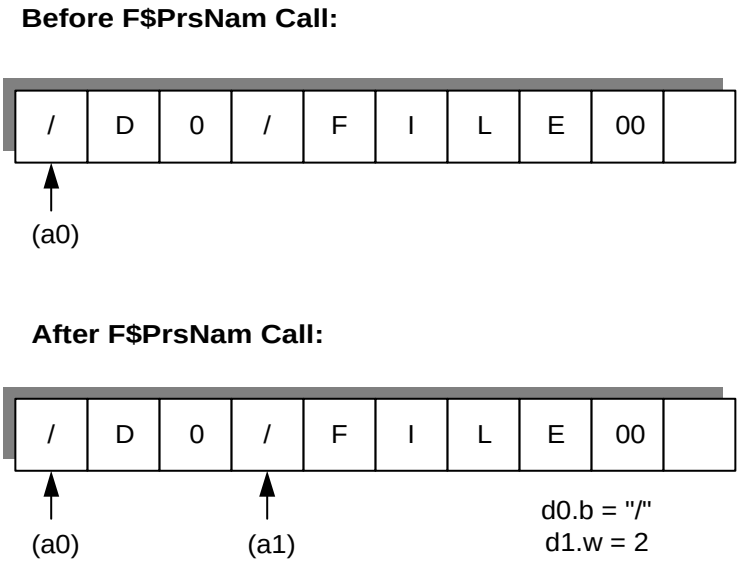
A - ZUppercase letters.	Periods
a - zLowercase letters _	Underscores
0 - 9Numbers$	Dollar signs

Any other character terminates the name and returns as the pathlist delimiter.

Note

F$PrsNam processes only one name, so you may need several calls to process a pathlist with more than one name. F$PrsNam terminates a name on recognizing a delimiter character. Pathlists are usually terminated with a null byte.

Figure D-5



Possible Errors

E$BNam

See Also

F$CmpNam

F$RetPD — Return Process/Path Descriptor

ASM Call

```
OS9 F$RetPD
```

Input

```
d0.w = process/path number
(a0) = process/path table pointer
```

Output

None

Error Output

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

State

☐ User ☒ System ☐ I/O

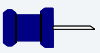
Description

F$RetPD de-allocates a process or path descriptor. It can be used with F$AllPD and F$FindPD to perform simple memory management of other fixed length objects.

See Also

F$AllPD

F$FindPD



Note

This is a privileged system-state service request.

F$RTE — Return from Interrupt Exception

ASM Call

```
OS9  F$RTE
```

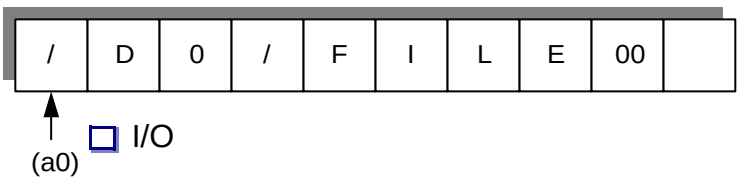
Input

None

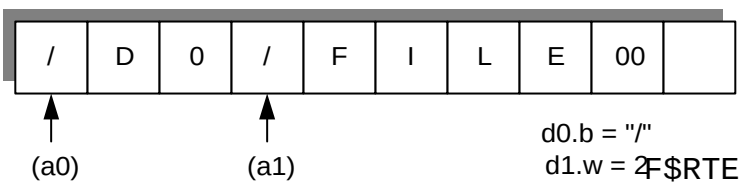
Output

None

Before F$PrsNam Call:



After F$PrsNam Call:



State

- [x] User
- [ ] System

Description

F$RTE may be used to exit from a signal processing routine.

F$RTE terminates a process signal intercept routine and continues executing the main program. However, if there are unprocessed signals pending, the interrupt routine executes again (until the queue is exhausted) before returning to the main program.

Note

When a signal is received, 70 bytes are used on the user stack. Consequently, intercept routines should be kept very short and fast if many signals are expected.

See Also

F$Icpt

F$SigReset

F$SchBit — Search Bit Map for Free Area

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

`F$SchBit` searches the specified allocation bit map for a free block (cleared bits) of the required length, starting at the beginning bit number (`d0.w`). `F$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.

Note

The IOMan module implements `F$SchBit`.

See Also

`F$AllBit`

`F$DelBit`

F$Send — Send Signal to Another Process

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$Send sends a signal to a specific process. The signal code is a word value. A process may send the same signal to multiple processes of the same group/user ID by passing 0 as the receiver's process ID number. For example, the OS-9 Shell command, `kill 0`, unconditionally aborts all processes with the same group/user ID (except the Shell itself). This is a handy but dangerous tool to get rid of unwanted background tasks.

If you attempt to send a signal to a process that has an unprocessed, previous signal pending, the signal is placed in a FIFO queue of signals for the individual process. If the process is in the signal intercept routine when it receives a signal, the new signal is processed when F$RTE executes.

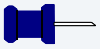
If the destination process for the signal is sleeping or waiting, it is activated so it may process the signal. The signal processing intercept routine is executed, if it exists (see F$Icpt). Otherwise, the signal aborts the destination process, and the signal code becomes the exit status (see F$Wait).

An exception is the wakeup signal. It activates a sleeping process but does not cause examination of the signal intercept routine and does 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
S$HangUp = 4 = Modem Hangup
        5-31 = Reserved for Microware; deadly to I/O
      32-255 = Reserved for Microware
   256-65535 = User defined
```

The S$Kill signal may only be sent to processes with the same group ID as the sender. Super users may kill any process.



Note

The I/O system uses the S$Wake signal extensively. It is not reliable if used by user-state programs.

Signal values less than 32 (S$Deadly) usually cause the current I/O operation to terminate with an error status equal to the signal value.

Possible Errors

E$IPrcID

See Also

F$Icpt

F$Sleep

F$Wait

F$SetCRC — Generate Valid CRC in Module

ASM Call

```
OS9 F$SetCRC
```

Input

```
(a0) = module pointer
```

Output

None

Error Output

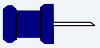
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

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 subsequently written to a file. The module must have correct size and sync bytes; other parts of the module are not checked.



Note

The module image must start on an even address or an address error occurs.

OS-9 does not permit any modification to the header of a module known to the system. Modifying the header makes the module inaccessible to other processes.

Possible Errors

E$BMID

See Also

F$CRC

F$SetSys — Set/Examine OS-9 System Global Variables

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$SetSys changes or examines 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.

Note

Only a super-user can change system variables. Any system variable may be examined, but only a few may be altered. The only useful variables that may be changed are `D_MinPty` and `D_MaxAge`. Consult **Process Scheduling** in Chapter 2: The Kernel for an explanation of what these variables control.

The system global variables are OS-9's data area. It is highly likely they will change from one release to another. You will probably have to relink programs using this system call to run them on future versions of OS-9.

WARNING

The super-user must be extremely careful when changing system global variables.

See Also

`F$SPrior` and the DEFS Files section in the ***OS-9 for 68K Processors Technical I/O Manual***.

F$Sigmask — Mask/Unmask Signals During Critical Code

ASM Call

```
OS9 F$SigMask
```

Input

```
d0.l = reserved, must be zero
d1.l = process signal level
        0 = clear
        1 = set/increment
       -1 = decrement
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

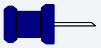
Description

F$SigMask enables or disables signals from reaching the calling process. If a signal is sent to a process whose mask is disabled, the signal is queued until the process mask becomes enabled. The process' signal intercept routine is executed with signals inherently masked.

Two exceptions to this rule are the S$Kill and S$Wake signals:

S$Kill	terminates the receiving process, regardless of the state of its mask.
S$Wake	ensures the process is active, but does not queue.

When a process makes an F$Sleep or F$Wait system call, its signal mask is automatically cleared. If a signal is already queued, these calls return immediately (to the intercept routine).



Note

Signals are analogous to hardware interrupts. Mask signals sparingly, and keep intercept routines as short and fast as possible.

F$SigReset — Reset Signal Intercept Context Stack

ASM Call

```
OS9 F$SigReset
```

Input

None

Output

None

Error Output

None

State

☒ User ☐ System ☐ I/O

Description

Under normal circumstances, OS-9 keeps the state of the main process on the system stack while a signal intercept routine executes. However, if signals are unmasked during the intercept routine, a subsequent signal causes the current state to be stacked on the user's stack.

This does not happen in simple cases, but if the intercept routine unmasks signals or `longjmps()`s and then unmasks signals, states are placed on the user's stack. There is no functional difference, and if the code actually expects to return through the nested intercept routines with multiple F$RTEs, the states must be left where they are.

If the code uses `longjmp()` to leave the intercept routine it implicitly clears the saved context off the stack. The kernel performs best if the code tells the kernel to remove the context.

Whenever a program `longjmp()`s out of an intercept routine or unmasks signals in an interrupt service routine with the intent of never F$RTEing, it should use `F$SigReset`:

```
if(setjmp(x) != 0) {
    _os_sigreset();
    _os_sigmask(-1);
}
```

See Also

F$Icpt

F$RTE

F$Sleep — Put Calling Process to Sleep

ASM Call

```
OS9 F$Sleep
```

Input

```
d0.l = Ticks/seconds (length of time to sleep)
```

Output

```
d0.l = Remaining number of ticks if awakened
       prematurely
```

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

F$Sleep deactivates the calling process until the requested number of ticks have elapsed.

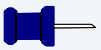
`Sleep(0)`	sleeps indefinitely.
`Sleep(1)`	gives up a time slice but does not necessarily sleep for one tick.

You cannot use F$Sleep to time more accurately than ±1 tick, because it is not known when the F$Sleep request was made during the current 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 resumes 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 resumes execution when it reaches the front of the queue. The process is activated before the full time interval if a signal (in particular S$Wake) is received. Sleeping indefinitely is a good way to wait for a signal or interrupt without wasting CPU time.

The duration of a tick is system dependent, but is usually .01 seconds. If the high order bit of d0.l is set, the low 31 bits are converted from 256ths of a second into ticks before sleeping to allow program delays to be independent of the system's clock rate.



Note

The system clock must be running to perform a timed sleep. It is not required to perform an indefinite sleep or to give up a time-slice.

Possible Errors

E$NoClk

See Also

F$Send

F$Wait

F$SPrior — Set Process Priority

ASM Call

```
OS9 F$SPrior
```

Input

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

Output

None

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

`F$SPrior` changes the process priority to the new value specified. 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 `0`), that may alter any process' priority.

Two system global variables affect task-switching:

`D_MinPty`	is the minimum priority 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.

Note

A very small change in relative priorities has a large effect. For example, if two processes have priorities 100 and 200, the process with the higher priority runs 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.

Possible Errors

`E$IPrcID`

See Also

`F$SetSys`

Process Scheduling in Chapter 2: The Kernel.

F$SRqCMem — System Request for Colored Memory

ASM Call

```
OS9 F$SRqCMem
```

Input

```
d0.l = Byte count of requested memory
d1.w = Memory type code (0 = any)
```

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
```

State

☒ User ☐ System ☐ I/O

Description

F$SRqCMem allocates a block of a specific type of memory. If a non-zero type is requested, the search is restricted to memory areas of that type. The area with the highest priority is searched first.

When the type code is 0, the search is based only on priority. This allows you to configure a system so fast, on-board memory is allocated before slow off-board memory. Areas with a priority of 0 are excluded from the search.

If more than one memory area has the same priority, the area with the largest total free space is searched first. This allows memory areas to be balanced (contain approximately equal amounts of free space).

Memory types or ***color codes*** are system dependent and may be arbitrarily assigned by the system administrator. Values below 256 are reserved for Microware use.

The requested number of bytes are rounded up to a system defined blocksize—currently 16 bytes. The memory always begins on an even boundary.

If -1 is passed in `d0.l`, the largest block of free memory of the specified type is allocated to the calling process.

`F$SRqMem` is equivalent to a `F$SRqCMem` request with a color of `0`.

Possible Errors

`E$Damage`

`E$MemFul`

`E$NoRAM`

See Also

`F$Mem`

`F$SRqMem`

`F$SRtMem`

Init: The Configuration Module and **Colored Memory** sections in Chapter 2: The Kernel

F$SRqMem — System Memory Request

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$SRqMem allocates a block of memory from the top of available RAM. The requested number of bytes is rounded up to a system defined blocksize (currently 16 bytes). This system call is useful for allocating I/O buffers and any other semi-permanent memory. The memory always begins on an even boundary.

If -1 is passed in d0.l, the largest block of free memory is allocated to the calling process.

Note

You ***must*** save the byte count of allocated memory (as well as the pointer to the block allocated) if the memory is ever to be returned to the system.

Possible Errors

E$MemFul

E$NoRAM

See Also

F$Mem

F$SRtMem

F$SRtMem — Return System Memory

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$SRtMem de-allocates memory after it is no longer needed. The number of bytes returned is rounded up to a system defined blocksize before the memory is returned. Rounding occurs identically to that done by F$SRqMem.

In user state, the system keeps track of memory allocated to a process and all blocks not returned are automatically de-allocated by the system when a process terminates. In system state, the process must explicitly return its memory.

Possible Errors

E$BPAddr

See Also

F$Mem

F$SRqMem

F$SSpd — Suspend Process

ASM Call

```
OS9 F$SSpd
```

Input

```
d0.w = process ID to suspend
```

Output

None

Error Output

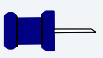
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

F$SSpd is currently not implemented.



Note

You can suspend a process by setting its priority below the system's minimum executable priority level (D_SysMin).

See Also

F$SetPri

F$SetSys

F$SSvc — Service Request Table Initialization

ASM Call

```
OS9 F$SSvc
```

Input

```
(a1) = pointer to service request initialization
       table
(a3) = user defined
```

Output

None

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

F$SSvc adds or replaces function requests in OS-9's user and privileged system service request tables.

(a3) is intended to point to global static storage. This allows a global data pointer to be associated with each installed system call. When the system call is invoked, the data pointer is automatically passed. Whatever (a3) points to is passed to the system call; (a3) may point to anything.

An example initialization table might look like this:

```
SvcTbl
  dc.w F$Service        OS-9 service request code
  dc.w Routine-*-2      Offset of routine to process request
```

```
    :
dc.w F$Service+SysTrap Redefine system level request
dc.w SysRoutn-*-2      Offset of routine to handle system
                       request
    :
dc.w -1 end of table
```

Valid service request codes range from (0-255).

Note

The offset desired is the offset from the beginning of the table entry to the routine minus 4. The minus 4 is to counteract incrementing done in the kernel.

If the sign bit of the function code word is set, only the system table is updated. Otherwise, both the system and user tables are updated.

You can only call privileged system service requests from routines executing in system (supervisor) state. The example above shows how a service call that must behave differently in system state than it does in user state is installed.

System service routines are executed in supervisor state. They are written to conform to register conventions:

Input:

```
d0-d4 = user’s values
(a0)-(a2) = user’s values
(a3) = service routine global data pointer
(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 carry bit is returned set, the service dispatcher sets `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 frame by the service routine.

Note

This is a privileged system-state service request.

F$STime — Set System Date and Time

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

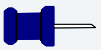
F$STime sets the current system date/time and starts the system real-time clock to produce time-slice interrupts. F$STime is accomplished by putting the date/time packet in the system direct storage area, and then linking the clock module. The clock initialization routine is called if the link is successful.

It is the function of the clock module to:

- Set up any hardware dependent functions to produce system tick interrupts (including moving new date/time into hardware, if needed).
- Install a service routine to clear the interrupt when a tick occurs.

The OS-9 kernel keeps 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.

The kernel uses the battery-backed clock format to call `F$STime` during system start-up. On systems with a battery-backed clock, this results in the system starting with the correct time and time-slicing turned on. If the system does not have a battery-backed clock (or if the `Init` module has disabled this feature), you should perform an `F$STime` call during the system start-up. For example, call `F$STime` in the initial application or system start-up file.



Note

The date and time are not checked for validity. On systems with a battery-backed clock, it is usually only necessary to supply the year to the `F$STime` call. The actual date and time are read from the real-time clock. To read the time, the month field in the date parameter must be `0`.

See Also

`F$Link`

`F$Time`

F$STrap — Set Error Trap Handler

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$STrap enters process local error trap routine(s) into the process descriptor dispatch table. If an entry for a particular routine already exists, it is replaced.

User programs may catch the following exception errors:

```
Bus errorAddress error
Illegal instructionZero Divide
CHK instructionTRAPV instruction
Privilege violationLine 1010 emulator
Line 1111 emulator
```

User programs can also catch the following exception errors on systems with a floating point coprocessor (68020 or 68030 with 68881/882; or 68040):

```
Branch or set on unordered condition
Inexact resultDivide by zero
UnderflowOperand Error
OverflowNAN signaled
```

If a user routine is not provided and one of these exceptions occur, 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 is passed the following:

```
d7.w = Exception vector offset
(a0) = Program counter when exception occurred
       (same as R$PC(a5))
(a1) = Stack pointer when exception occurred
       (R$a7(a5))
(a5) = User's register stack image when exception
       occurred
(a6) = user's primary global data pointer
```

To return to normal program execution after handling the error, the exception must restore all registers (from the register image at `(a5)`) and jump to the return program counter. For some kinds of exceptions (especially bus and address errors), this may not be appropriate. It is the user program's responsibility to determine whether and where to continue execution.

You can disable an error exception handler by calling `F$STrap` with an initialization table specifying zero as the offset to the routine(s) to remove. For example, the following table removes user routines for the `trapv` and `chk` error exceptions:

```
Table     dc.w  T_TRAPV, 0
          dc.w  T_CHK, 0
          dc.w  -1
```

Note

Beware of exceptions in exception handling routines. They are usually not re-entrant.

F$SUser — Set User ID Number

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$SUser alters the current user ID to the specified ID. F$SUser has the following restrictions:

- User number 0.0 may change their ID to anything without restriction.
- A primary module owned by user 0.0 may change its ID to anything without restriction.
- Any primary module may change its user ID to match the module's owner.

All other attempts to change user ID number return an E$Permit error.

F$SysDbg — Call System Debugger

ASM Call

```
OS9 F$SysDbg
```

Input

None

Output

None

Error Output

```
cc = Carry set
d1.w = Appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

F$SysDbg calls the system level debugger, if one exists, to debug system-state routines, such as device drivers. The system level debugger runs in system state and effectively stops timesharing when it is active. It should never be used when there are other users on the system.

Note

You must enable the system debugger before installing breakpoints or attempting to trace instructions. If no system debugger is available, the system is 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.

If F$SysDbg is called from user-state, the user process must have a group.user ID of 0.0. Calls from system-state (for example, file managers, device drivers) do not have this restriction.

F$SysID — Get System Information

ASM Call

```
OS9 F$SysID
```

Input

```
d0.l = reserved, must be zero.
(a0) = pointer to SysIdent buffer (or null)
(a1) = pointer to copyright buffer (or null)
(a2) = reserved, must be zero.
(a3) = reserved, must be zero.
```

Output

```
d0.l = kernel OEM registration number
d1.l = kernel copy serial number
d2.l = processor identifier (68000/68010, etc.)
d3.l = kernel (OS) identifier (68000/68010, etc.)
d4.l = floating pointer unit identifier
       (68881, 68882, 68040)
d5.l = reserved (0)
d6.l = kernel version
     (level/version/revision/edition)
d7.l = reserved (0)
(a0) = “OS-9 Version Vm.n” string (if non-zero)
       (SysIdent buffer)
(a1) = “Copyright....” string (if non-zero)
       (Copyright buffer)
(a2) = reserved
(a3) = reserved
```

Error Output

```
cc = carry bit set
d1.w = appropriate error code
```

State

☒ User ☐ System ☐ I/O

Description

F$SysID returns information about the system. You can use it to determine specific available operating system capabilities (such as the operating system's (OS) release level) or system hardware characteristics (such as floating point unit presence).

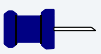
The processor identifier field indicates the class of processor in use, such as 68000, 68020, etc. CPU32 family processors are assigned a class of 68300.

The kernel (OS) identifier identifies the specific version kernel in use. This may differ from the processor identifier field whenever a specific version kernel is available for a class of processors. For example, the 68349 is a CPU32 (68300) processor family member.

The floating point unit identifier identifies the release level of the kernel in four 1-byte fields. Thus, Level 1, OS-9 for 68K version 3.0, Edition 0 would be a value of 01030000.

The SysIdent buffer returns a string of the form OS-9 Version Vm.n if the input is non-zero.

The copyright buffer returns a string of the form Copyright .... if the input (a1) is non-zero.



Note

Note the following:

- For the Atomic kernel, the string buffers (if non-zero) cause a NULL to be returned in the buffer.
- The string buffers, if used, must be 80 characters. The maximum strings returned are 79 characters plus a NULL terminator.

- Applications can check for ***OS-9 for 68K pre-version 3.n*** configurations of the kernel by performing this call and checking the kernel version field returned. If it is `0`, the kernel is ***OS-9 for 68K pre-version 3.n***.

F$Time — Get System Date and Time

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

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. You can use this to determine 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:

Figure D-6 Gregorian vs. Julian Time

	Register Offset		Gregorian Format	Julian Format
d0.l	byte	3	zero	seconds since midnight (long) (86399
		2	hour (0-23)	
		1	minute (0-59)	
		0	second (0-59)	
d1.l	byte	2-3	year (integer)	Julian day numb (long)
		1	month (1-12)	
		0	day (1-31)	

Note

`F$Time` returns a date and time of zero (with no error) if no previous call to `F$STime` is made. A tick rate of zero indicates the clock is not running.

See Also

`F$Julian`

`F$STime`

F$TLink — Install User Trap Handler Module

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

You can use user traps 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 using the library code if it is changed (without having to re-compile or re-link the program itself). Most Microware utilities use one or more trap libraries.

F$TLink 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 necessary. You can remove traps by passing a null pointer.

A user program calls a trap routine using the following assembly language directive:

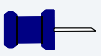
```
tcall N,Function
```

This is the equivalent to:

```
trap #N
dc.w Function
```

N can be 1 to 15 (specifying which user trap vector to use). OS-9 does not use the function code, except for passing it to the trap handler, and the program counter is skipped past it.

F$TLink allows the program to delay installation of the handler until the program actually uses a trap. If a user program executes a user trap call before the corresponding F$TLink call has been made, the system executes the user's default trap exception entry point (specified in the module header) if one exists.



Note

System-state processes should not attempt to use trap handlers.

See Also

Chapter 5: User Trap Handlers

F$Trans — Translate Memory Address

ASM Call

```
OS9 F$Trans
```

Input

```
d0.l = size of block to translate
d1.l = mode: 0 - local CPU address to external bus
             addr
             1 - external bus address to local CPU
             addr
(a0) = address of block
```

Output

```
d0.l = size of block translated
(a0) = translated address of block
```

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

On systems with dual-ported memory, a memory location may appear at different addresses depending on whether it is accessed via the ***local*** CPU bus or the system's external bus. You can use the `F$Trans` request to translate an address to or from its external bus address.

`F$Trans` is used when the external bus address must be passed to hardware devices, such as DMA-type controllers. Using the local CPU bus address is faster and reduces the traffic on the external bus. Generally, you should only use the system's external bus address if you cannot use the local CPU bus address.

If the specified source block is non-linear with respect to its destination mapping, F$Trans returns the maximum number of bytes accessible at the translated address. In this case, subsequent calls to F$Trans must be made until the entire block has been successfully translated. This is rare, since OS-9's memory management routines do not allocate non-linear blocks.

Possible Errors

E$IBA

E$Param

E$UnkSvc

See Also

Chapter 2: The Kernel sections on**Init: The Configuration Module** and **Colored Memory**.

F$UAcct — User Accounting

ASM Call

```
OS9 F$UAcct
```

Input

```
d0.w = Function code (F$Fork, F$Chain, F$Exit)
(a0) = Process descriptor pointer
```

Output

None

Error Output

```
cc = carry bit set
d1.w = error code if error
```

State

☒ User ☐ System ☐ I/O

Description

F$UAcct is a user-defined system call that may be installed by an OS9P2 module. It is called in system state at the beginning and end of every process; whenever you execute F$Fork, F$Chain, or F$Exit.

The kernel's fork and chain routines make an F$UAcct request just before a new process is inserted in the active queue. Since the new process is ready to execute, its user number, priority, primary module, parameters, etc. are known to F$UAcct. This provides a variety of opportunities for a F$UAcct routine. For example:

- A system administrator could keep track of every program run and who ran what program.
- Automatically lower the priority of particular programs.
- Keep a log of everything a specific user does.

Note

If F$UAcct returns an error during F$Fork, the new process terminates with the error code in d1.w.

OS-9's process termination routine makes a F$UAcct request just before a process' resources are returned to the system. The process descriptor contains information about the CPU time consumed, how many bytes were read or written, how many system calls were made, etc. Once again, F$UAcct could be used to record or react to this information. The system ignores any F$UAcct error returned at the end of a process.

Note

You must preserve the values in all registers except d0 and d1.

Possible Errors

E$Param

E$UnkSvc

See Also

F$SSvc

Chapter 2: The Kernel (section on installing system-state routines).

F$UnLink — Unlink Module by Address

ASM Call

```
OS9 F$UnLink
```

Input

```
(a2) = Address of the module header
```

Output

None

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

F$UnLink tells OS-9 the module is no longer needed by the calling process. The module's link count is decremented. When the link count is zero, the module is removed from the module directory and its memory is de-allocated. 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.

See Also

F$UnLoad

F$UnLoad — Unlink Module by Name

ASM 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
```

State

☒ User ☐ System ☐ I/O

Description

F$UnLoad locates the module in the module directory, decrements its link count, and removes it from the directory if the count reaches zero. Note this call differs from F$UnLink as the pointer to the module name is supplied rather than the address of the module header.

See Also

F$UnLink

F$VModul — Validate Module

ASM Call

```
OS9 F$VModul
```

Input

```
d0.l = beginning of module group (ID)
d1.l = module size
(a0) = module pointer
```

Output

```
(a2) = Directory entry pointer
```

Error Output

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

State

☐ User ☒ System ☐ I/O

Description

`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.

Possible Errors

E$BMCRC

E$BMID

E$BMHP

E$DirFul

E$KwnMod

See Also

F$CRC

F$Load

Note

This is a privileged system-state service request.

F$Wait — Wait for Child Process to Terminate

ASM Call

```
OS9 F$Wait
```

Input

None

Output

```
d0.w = Terminating child process's ID
d1.w = Child process's exit status code
```

Error Output

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

State

☒ User ☐ System ☐ I/O

Description

F$Wait causes the calling process to deactivate until a child process terminates by executing a F$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 (register d1) is the signal code.

If the caller has several child processes, 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 process died before the Wait call, the caller is reactivated immediately. Wait returns an error only if the caller has no child processes.

Note

The process descriptors for child processes are not returned to free memory until their parent process does a `F$Wait` system call or terminates.

If a signal is received by a process waiting for children to terminate, it is activated. In this case, `d0.w` contains zero, since no child process has terminated.

Possible Errors

`E$NoChld`

See Also

`F$Exit`

`F$Fork`

`F$Send`

I$Attach — Attach New Device to System

ASM Call

```
OS9 I$Attach
```

Input

```
d0.b = Access mode (Read_, Write_, Updat_)
(a0) = Device name pointer
```

Output

```
(a2) = Address of the device table entry
```

Error Output

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

State

☒ User ☒ System ☒ I/O

Description

I$Attach causes an I/O device to become known to the system. Use it to attach a new device to the system, or to verify 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 by which the device is known). The descriptor module contains the name of the device's:

- File manager
- Device driver
- Other related information

If the descriptor is found and the device is not already attached, OS-9 links 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 is not re-initialized.

You can use the access mode parameter to verify that subsequent read and/or write operations are permitted. An `Attach` system call is not required to perform routine I/O. It does not reserve the device in question; `I$Attach` simply prepares it for subsequent use by any process.

The kernel attaches all devices at open and detaches them at close.

Note

The IOMan module implements `I$Attach`.

Note

`Attach` and `Detach` for devices are similar to `Link` and `Unlink` for modules; they are usually used together. However, system performance can improve slightly if all devices are attached at startup. This increments each device's use count and prevents the device from being re-initialized every time it is opened. This also has the advantage of allocating the static storage for devices all at once, thus minimizing memory fragmentation. If this is done, the device driver termination routine is never executed.

Possible Errors

`E$BMode`

`E$DevBsy`

`E$DevOvf`

`E$MemFul`

See Also

I$Detach

I$ChgDir — Change Working Directory

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

`I$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 change. 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

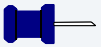
3 = Update (read and write)

4 = Execute

If the access mode is read, write, or update, the current data directory changes. If the access mode is execute, the current execution directory changes. Both can change simultaneously.

Note

The shell `chd` directive uses `Update` mode—you must have both read and write permission to change directories from the shell. This is a recommended practice.



Note

The IOMan module implements `I$ChgDir`.

Possible Errors

`E$BMode`

`E$BPNam`

I$Close — Close Path to File/Device

ASM Call

```
OS9 I$Close
```

Input

```
d0.w = Path number
```

Output

None

Error Output

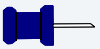
```
cc = Carry bit set
d1.w = Appropriate error code
```

State

☒ User ☒ System ☒ I/O

Description

`I$Close` terminates the I/O path specified by the path. The path number is no longer valid for any OS-9 calls unless it becomes active again through an `Open`, `Create`, or `Dup` system call. When pathlists to non-sharable devices are closed, the devices become available to other requesting processes. If this is the last use of the path (it has not been inherited or duplicated by `I$Dup`), all OS-9 internally managed buffers and descriptors are de-allocated.

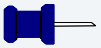


Note

The OS-9 `F$Exit` service request automatically closes any open paths. By convention, standard I/O paths are not closed unless it is necessary to change the files/devices they correspond to.

Note

`I$Close` does an implied `I$Detach` call. If this causes the device use count to become zero, the device termination routine is executed.



Note

The IOMan module implements `I$Close`.

Possible Errors

`E$BPNum`

See Also

`I$Detach`

I$Create — Create Path to New File

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

`I$Create` creates a new file. On multi-file devices, the new file name is entered in the directory structure, and `Create` is synonymous with `Open`.

The access mode parameter passed in register d0.b must have the write bit set if any data is to be written to the file. The file is given the attributes passed in the register d1.w. The individual bits are defined as follows:

Table D-11 I$Create Bits

Mode Bits (d0.w)	Attribute Bits (d1.w)
0 = Read	0 = Owner Read Permission
1 = Write	1 = Owner Write Permission
2 = Execute	2 = Owner Execute Permission
5 = Initial File Size	3 = Public Read Permission
6 = Single User	4 = Public Write Permission
	5 = Public Execute Permission
	6 = Non-Sharable File

If the execute bit (bit 2) of the access mode byte is set, directory searching begins with the working execution directory instead of the working data directory.

The path number returned by OS-9 identifies the file in subsequent I/O service requests until the file is closed.

Write automatically allocates file space for the file. The SetStat call (SS_Size) explicitly allocates file space. If the size bit (bit 5) is set, an initial file size estimate may be passed in d2.l.

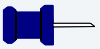
An error occurs if the pathlist specifies an already existing file name. You cannot use I$Create to make directory files (see I$MakDir).

Create causes an implicit I$Attach call. If the device has not previously been attached, the device's initialization routine is executed.

Note

The caller is made the owner of the file. To maintain compatibility 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.

If an initial file size is specified with `I$Create`, the exact amount specified may not be allocated. You must execute a `SS_Size` SetStat after creating the file to ensure sufficient space was allocated.



Note

The IOMan module implements `I$Create`.

Possible Errors

`E$BPNam`

`E$PthFul`

See Also

`I$Attach`

`I$Close`

`I$MakDir`

`I$Open`

I$Delete — Delete File

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

I$Delete 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 results. An attempt to delete a non-multifile device results 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 if a full pathlist is specified (a pathlist beginning with a slash (/)), the access mode is ignored.

Note

The IOMan module implements `I$Delete`.

Possible Errors

`E$BPNam`

See Also

`I$Attach`

`I$Create`

`I$Detach`

`I$Open`

I$Detach — Remove Device from System

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

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 de-allocated. 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.

You must use the I$Detach service request to detach devices that were attached with the I$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 set up 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, and then detached to free system resources when no longer needed.

Note

If an invalid address is passed in `(a2)`, the system may crash or undergo severe damage.

Note

The IOMan module implements `I$Detach`.

See Also

`I$Attach`

`I$Close`

I$Dup — Duplicate Path

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

Given the number of an existing path, I$Dup returns a synonymous path number for the same file or device. I$Dup always uses the lowest available path number. For example, if you do I$Close on path #0, then do I$Dup on path #4, path #0 is returned as the new path number. In this way, the standard I/O paths may be manipulated to contain any desired paths.

The 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.

Note

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.

Note

The IOMan module implements `I$Dup`.

Possible Errors

`E$BPNum`

`E$PthFul`

I$GetStt — Get File/Device Status

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

This is a wildcard 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 (such as echo on/off and delete character). It is commonly used 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 0-255 are reserved for Microware use. The remaining codes and their parameter passing conventions are user definable (see Chapter 3, the section on device drivers).

Note

The IOMan module implements `I$GetStt`.

Presently defined `I$GetStt` function codes are listed below:

Table D-12 `I$GetStt` Function Codes

Name	Description
`SS_CDFD`	Return File Descriptor (CDFM)
`SS_DevNam`	Return Device Name (All)
`SS_EOF`	Test of End of File (RBF, SCF, PIPE)
`SS_FD`	Read File Descriptor Sector (RBF, PIPE)
`SS_FDInf`	Get Specified File Descriptor Sector (RBF)
`SS_Free`	Return Amount of Free Space on Device (NRF, NVRAM file mgr.)
`SS_Opt`	Read `PD_OPT`: The Path Descriptor Option Section (All)
`SS_Pos`	Get Current File Position (RBF, PIPE)
`SS_Ready`	Test for Data Ready (RBF, SCF, PIPE)
`SS_Size`	Return Current File Size (RBF, PIPE)
`SS_VarSect`	Query Support for Variable Logical Sector Sizes (RBF)

Possible Errors

E$BPNum

SS_CDFD — Return File Descriptor (CDFM)

Input

```
d0.w = Path number
d1.w = #SS_CDFD function code
d2.w = Number of bytes to copy
(a0) = Pointer to buffer area for file descriptor
```

Output

None

Error Output

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

State

☒ User ☒ System ☒ I/O

Description

SS_CDFD reads the file descriptor describing the path number. The file descriptor may be read for information purposes only, as there are no user changeable parameters.

SS_DevNam — Return Device Name (ALL)

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.

State

☒ User ☒ System ☒ I/O

SS_E0F

Test for End of File (RBF, SCF, PIPE)

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)
```

State

☒ User ☒ System ☒ I/O

SS_FD — Read File Descriptor Sector (RBF, PIPE)

Input

```
d0.w = Path number
d1.w = #SS_FD function code
d2.w = Number of bytes to copy(<=logical sector size
       of media)
(a0) = Address of buffer area for FD
```

Output

File descriptor copied into buffer

State

☒ User ☒ System ☒ I/O

Description

Use SS_FD to inspect the file descriptor information (for example, FD_OWN and FD_DAT) and the file segment list.

SS_FDInf — Get Specified File Descriptor Sector (RBF)

Input

```
d0.w = Path number
d1.w = #SS_FDInf function code
d2.w = Number of bytes to copy (<=256)
d3.l = FD sector address
(a0) = Address of buffer area for FD
```

Output

File descriptor copied into buffer.

State

☒ User ☒ System ☒ I/O

Note

If `SS_FDInf` is called in user state, the caller must be a super-group user. If it is called in system state, the caller does *not* have to be a super-group user.

SS_Free

Return Amount of Free Space on Device (NRF, NVRAM file mgr.)

Input

```
d0.l = Path number
d1.w = #SS_Free function code
```

Output

```
d0.l = Size of free space on device, in bytes
```

State

☒ User ☒ System ☒ I/O

SS_Opt

Read PD_OPT: The Path Descriptor Option Section (All)

Input

```
d0.w = Path number
d1.w = #SS_Opt function code
(a0) = Address to put a 128 byte status packet
```

Output

Status packet copied to buffer.

Error Output

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

State

☒ User ☒ System ☒ I/O

Description

`SS_Opt` 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, refer to Chapter 3: OS-9 Input/Output System the section on file manager path descriptors.

SS_Pos — Get Current File Position (RBF, PIPE)

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
```

State

☒ User ☒ System ☒ I/O

SS_Ready — Test for Data Ready (RBF, SCF, PIPE)

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$NotRdy if no data is
       available)
```

State

☒ User ☒ System ☒ I/O

SS_Size — Return Current File Size (RBF, PIPE)

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
```

State

☒ User ☒ System ☒ I/O

SS_VarSect

Query Support for Variable Logical Sector Sizes (RBF)

Input

```
d0.w = path number
d1.w = #SS_VarSect function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_VarSect is an internal call between RBF and a driver. If the driver does not return an error, the logical sector size of the media is specified in PD_SSize. If the driver returns an error, and the error is E$UnkSvc, RBF sets the path's logical sector size to 256 bytes and ignores PD_SSize. If any other error is returned, the path open is aborted and the error is returned to the caller.

I$MakDir — Make New Directory

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

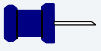
`I$MakDir` is the only way to create a new directory file. It creates and initializes a new directory as specified by the pathlist. The new directory file contains no entries, except for an entry for itself (specified by a dot (`.`)) and its parent directory (specified by double dot (`..`)). `I$MakDir` fails on non-multi-file devices. If the execution bit is set, OS-9 begins searching for the file in the working execution directory (unless the pathlist begins with a slash).

The caller is made the owner of the directory. `I$MakDir` does not return a path number because directory files are not opened by this request (use `I$Open` to do so). The new directory automatically has its directory bit set in the access permission attributes. The remaining

attributes are specified by the bytes passed in register `d1.w` that have individual bits defined as listed below (if the bit is set, access is permitted):

Table D-13 `I$MakDir` Bits

Mode Bits (d0.b)	Attribute Bits (d1.w)
`0` = Read	`0` = Owner Read Permission
`1` = Write	`1` = Owner Write Permission
`2` = Execute	`2` = Owner Execute Permission
`5` = Initial Directory Size	`3` = Public Read Permission
`7` = Directory	`4` = Public Write Permission
	`5` = Public Execute Permission
	`6` = Non-Sharable File
	`7` = Directory



Note

The IOMan module implements `I$MakDir`.

Possible Errors

`E$BPNam`

`E$CEF`

`E$PNNF`

I$Open — Open a Path to a File or Device

ASM Call

```
OS9 I$Open
```

Input

```
d0.b = Access mode (D S E W R)
(a0) = Pathname pointer
```

Output

```
d0.w = Path number
(a0) = Updated past pathname
```

Error Output

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

State

☒ User ☒ System ☒ I/O

Description

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):

Table D-14 `I$Open` Bits

Mode Bits
`0` = Read
`1` = Write
`2` = Execute
`4` = Force writes to always append to end-of-file
`6` = Open File for Non-Sharable Use
`7` = Open Directory File

Note

A non-directory file may be opened with no bits set. This allows you to examine the attributes, size, etc. with the `GetStt` system call, but does not permit any actual I/O on the path.

For RBF devices, use read mode instead of update if you are not going to modify the file. 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`).

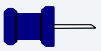
If the execution bit mode is set, OS-9 begins searching for the file in the working execution directory (unless the pathlist begins with a slash).

If the single user bit is set, the file is opened for non-sharable access even if the file is sharable.

If the append user bit is set, all writes to the file are written at end-of-file.

Files can be opened by several processes (users) simultaneously. Devices have an attribute specifying whether or not they are sharable on an individual basis.

I$Open always uses the lowest path number available for the process.



Note

The IOMan module implements I$Open.

Directory files may be opened only if the directory bit (bit 7) is set in the access mode.

Possible Errors

E$Bmode

E$BPNam

E$FNA

E$PNNF

E$PthFul

E$Share

See Also

I$Attach

I$Close

I$Create

I$Read

Read Data from File or Device

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

I$Read reads a specified number of bytes from the specified path number. 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 and line delete. If there is not enough data in the file to satisfy the read request, fewer bytes are 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 returns an end of file error.

Note

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 are set to zero. You may wish to modify the device descriptor so 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.

The IOMan module implements `I$Read`.

For RBF devices, if the file is open for update, the record read is locked out. See the Record Locking section in the RBF chapter of the ***OS-9 for 68K Processors Technical I/O Manual***.

The number of bytes requested is read unless:

- The end-of-file is reached
- An end-of-record occurs (SCF only)
- An error condition occurs

Possible Errors

`E$BMode`

`E$BPNum`

`E$EOF`

`E$Read`

See Also

`I$ReadLn`

I$ReadLn — Read Text Line with Editing

ASM 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
```

State

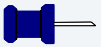
☒ User ☒ System ☒ I/O

Description

I$ReadLn is similar to I$Read except it reads data from the input file or device until an end-of-line character is encountered. I$ReadLn also causes line editing to occur on SCF-type devices. Line editing refers to backspace, line delete, echo, automatic line feed, etc. Some devices (SCF) may limit the number of bytes read with one call.

SCF requires the last byte entered be an end-of-record character (normally carriage return). If more data is entered than the maximum specified, it is not accepted and a PD_OVF character (normally bell) is echoed. For example, a I$ReadLn of exactly one byte accepts only a carriage return to return without error and beeps when other keys are pressed.

After all data in a file has been read, the next I$ReadLn service request returns an end of file error.



Note

The IOMan module implements I$ReadLn.

Possible Errors

E$BMode

E$BPNum

E$Read

See Also

I$Read

I$Seek

Reposition Logical File Pointer

ASM 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
```

State

☒ User ☒ System ☒ I/O

Description

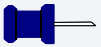
`I$Seek` repositions the path's file pointer which is the 32-bit address of the next byte in the file to be read or written. `I$Seek` usually does not initiate any physical positioning of the media.

You can perform a `Seek` to any value even if the file is not large enough. Subsequent writes automatically expand the file to the required size (if possible), but reads return an end-of-file condition.

Note

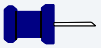
A `Seek` to address zero is the same as a rewind operation.

Seeks to non-random access devices are usually ignored and return without error.



Note

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 takes negatives as large positive numbers.



Note

The IOMan module implements `I$Seek`.

Possible Errors

`E$BPNum`

I$SetStt — Set File/Device Status

ASM 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
```

State

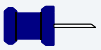
☒ User ☒ System ☒ I/O

Description

This is a ***wildcard*** 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 with the I$GetStt service request that reads the device's operating parameters.

The mnemonics for the status codes are found in the relocatable library `sys.l` or `usr.l`. Codes 0-255 are reserved for Microware use. The remaining codes and their parameter passing conventions are user definable (see Chapter 3: OS-9 Input/Output System, the **Device Driver Overview** section).



Note

The IOMan module implements `I$SetStt`.

Presently defined `I$SetStt` function codes are listed below:

Table D-15 `I$SetStt` Function Codes

Name	Description
`SS_Attr`	Set File Attributes (RBF, PIPE)
`SS_Close`	Notify Driver That Path Has Been Closed (SCF, RBF, SBF)
`SS_DCOff`	Send Signal When Data Carrier Detect Line Goes False (SCF)
`SS_DCOn`	Send Signal When Data Carrier Detect Line Goes True (SCF)
`SS_DsRTS`	Disable RTS Line (SCF)
`SS_EnRTS`	Enable RTS Line (SCF)
`SS_Feed`	Erase Tape (SBF)
`SS_FD`	Write File Descriptor Sector (RBF)
`SS_Lock`	Lock Out Record (RBF)

Table D-15 I$SetStt Function Codes (continued)

Name	Description
SS_Open	Notify Driver That Path Has Been Opened
SS_Opt	Write Option Section of Path Descriptor (All)
SS_Relea	Release Device (SCF, PIPE)
SS_Reset	Restore Head to Track Zero (RBF, SBF)
SS_RFM	Skip Tape Marks (SBF)
SS_Size	Set File Size (RBF, PIPE)
SS_Skip	Skip Blocks (SBF)
SS_SSig	Send Signal on Data Ready (SCF, PIPE)
SS_Ticks	Wait Specified Number of Ticks for Record Release (RBF)
SS_WFM	Write Tape Marks (SBF)
SS_WTrk	Write (Format) Track (RBF)

Possible Errors

E$BPNum

SS_Attr — Set File Attributes (RBF, PIPE)

Input

```
d0.w = Path number
d1.w = #SS_Attr function code
d2.b = New attributes
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Attr 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_Close

Notify Driver That Path Has Been Closed (SCF, RBF, SBF)

Input

```
d0.w = path number
d1.w = SS_Close function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Close is an internal call for drivers.

SS_DCOff — Send Signal When Data Carrier Detect Line Goes False (SCF)

Input

```
d0.w = path number
d1.w = SS_DCOff function code
d2.w = Signal code to be sent
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

When a modem has finished receiving data from a carrier, the data carrier detect line goes false. SS_DCOff sends a signal code when this happens. SS_DCOn sends a signal when the line goes true.

SS_DCOn

Send Signal When Data Carrier Detect Line Goes True (SCF)

Input

```
d0.w = path number
d1.w = SS_DCOn function code
d2.w = Signal code to be sent
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

When a modem receives a carrier, the data carrier detect line goes true. SS_DCOn sends a signal code when this happens. SS_DCOff sends a signal when the line goes false.

SS_DsRTS — Disable RTS Line (SCF)

Input

```
d0.w = path number
d1.w = SS_DsRTS function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_DsRTS tells the driver to negate the RTS hardware handshake line.

SS_EnRTS — Enable RTS Line (SCF)

Input

```
d0.w = path number
d1.w = SS_EnRTS function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_EnRTS tells the driver to enable the RTS hardware handshake line.

SS_Feed — Erase Tape (SBF)

Input

```
d0.w = path number
d1.w = SS_Feed function code
d2.l = # of blocks to erase
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Feed erases a portion of the tape. The amount of tape erased depends on the capabilities of the hardware used. SBF attempts to use the following: If -1 is passed in d2, SBF erases until the end-of-tape is reached. If d2 receives a positive parameter, SBF erases the amount of tape equivalent to that number of blocks. This depends on both the hardware used and the driver.

SS_FD — Write File Descriptor Sector (RBF)

Input

```
d0.w = Path Number
d1.w = #SS_FD function code
(a0) = Address of FD sector image
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_FD changes file descriptor sector data. The path must be open for write.

Note

You can only change FD_OWN, FD_DAT, and FD_Creat. These are the only fields written back to disk. Only the super user can change the file's owner ID.

SS_FD should normally be used with GetStat (SS_FD) to read the File Descriptor (FD) before attempting to change FD sector data.

SS_Lock

Lock Out Record (RBF)

Input

```
d0.w = Path Number
d1.w = #SS_Lock function code
d2.l = Lockout size
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

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 remaining 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_Open — Notify Driver That Path Has Been Opened

Input

```
d0.w = path number
d1.w = SS_Open function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Open is an internal call for drivers.

SS_Opt

Write Option Section of Path Descriptor (ALL)

Input

```
d0.w = Path number
d1.w = #SS_Opt function code
(a0) = Address of a 128 byte status packet
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

`SS_Opt` 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 (such as echo and auto line feed). This call is handled by the file managers, and only copies values that are appropriate to be changed by user programs.

SS_Relea — Release Device (SCF, PIPE)

Input

```
d0.w = path number
d1.w = SS_Relea function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Relea releases the device from any SS_SSig, SS_DCOn, or SS_DCOff requests made by the calling process on this path.

SS_Reset — Restore Head to Track Zero (RBF, SBF)

Input

```
d0.w = Path number
d1.w = #SS_Reset function code
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

For RBF, this directs the disk head to track zero. It is used for formatting and for error recovery. For SBF, this rewinds the tape.

SS_RFM — Skip Tape Marks (SBF)

Input

```
d0.w = path number
d1.w = SS_RFM function code
d2.l = # of tape marks
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_RFM skips the number of tape marks specified in d2. If d2 is negative, the tape is rewound the specified number of marks.

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

State

☒ User ☒ System ☒ I/O

Description

SS_Size sets the file's size.

For pipe files, you can use SS_Size to reset the pipe path (`d2.l=0`), provided the pipe has no active readers or writers. Any other value in `d2.l` is ignored.

SS_Skip

Skip Blocks (SBF)

Input

```
d0.w = path number
d1.w = SS_Skip function code
d2.l = # of blocks to skip
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_Skip skips the number of blocks specified in d2. If the number is negative, the tape is rewound the specified number of blocks.

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

State

☒ User ☒ System ☒ I/O

Description

SS_SSig sets up a signal to send to a process when an interactive device or pipe has data ready. SS_SSig must be reset each time the signal is sent. The device or pipe is considered busy and returns an error if any read request arrives before the signal is sent. Write requests to the device are allowed in this state.

SS_Ticks

Wait Specified Number of Ticks for Record Release (RBF)

Input

```
d0.w = path number
d1.w = #SS_Ticks function code
d2.l = Delay interval
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

Normally, if a read or write request is issued for a part of a file locked out by another user, RBF sleeps indefinitely until the conflict is removed.

You can use SS_Ticks to return an error (E$Lock) 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 0 (RBF's default) causes a sleep forever until the record is released. A value of 1 means if the record is not released immediately, an error is returned. If the high order bit is set, the lower 31 bits are converted from 256th of a second into ticks before sleeping. This allows programmed delays to be independent of the system clock rate.

SS_WFM — Write Tape Marks (SBF)

Input

```
d0.w = path number
d1.w = SS_WFM function code
d2.l = # of tape marks
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_WFM writes the number of tape marks specified in d2.

SS_WTrk — Write (Format) Track (RBF)

Input

```
d0.w = Path number
d1.w = #SS_WTrk function code
(a0) = Address of track buffer
       For hard disks and "autosize" media, this
       table contains 1 logical sector of data
       (pattern $E5). For floppy disks, this table
       contains the track's physical data.
(a1) = Address of interleave table
       This table contains byte entries of LSN's
       ordered to match the requested interleave
       offset. NOTE: This is a "logical" table and
       does not reflect the PD_SOffs base sector
       number.
d2   = Track number
d3.w = Side/density
      The low order byte has 3 bits which can be set:
           Bit 0 = SIDE (0=side zero;1=side one)
           Bit 1 = DENSITY (0=single;1=double)
           Bit 2 = TRACK DENSITY (0=single;1=double)
       The high order byte contains the side number.
d4   = Interleave value
```

Output

None

State

☒ User ☒ System ☒ I/O

Description

SS_WTrk causes a format track operation (used with most floppy disks) to occur. For hard or floppy disks with a ***format entire disk*** command, this formats the entire media only when side 0 of the first accessible track is specified.

I$Write — Write Data to File or Device

ASM 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
```

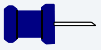
State

☒ User ☒ System ☒ I/O

Description

`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.



Note

The IOMan module implements `I$Write`.

On RBF devices, any locked record is released.

Possible Errors

E$BMode

E$BPNum

E$Write

See Also

I$Create

I$Open

I$WritLn

I$WritLn — Write Line of Text with Editing

ASM Call

```
OS9 I$WritLn
```

Input

```
d0.w = Path number
d1.l = Maximum number of bytes to write
(a0) = Address of buffer
```

Output

```
d1.l = Actual number of bytes written
```

Error Output

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

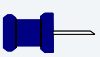
State

☒ User ☒ System ☒ I/O

Description

`I$WritLn` is similar to `I$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 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 are not counted.



Note

The IOMan module implements `I$WritLn`.

On RBF devices, any locked record is released.

Possible Errors

E$BMode

E$BPNum

E$Write

See Also

I$Create

I$Open

I$Write

OS-9 for 68K Processors Technical I/O Manual chapter on SCF Drivers (line editing).

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Product Discrepancy Report

To: Microware Customer Support

FAX: 515-224-1352

From:_____

Company:_____

Phone:_____

Fax:_____Email:_____

Product Name:

Description of Problem:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Host Platform_____

Target Platform_____

MICROWARE™