OS-9 Technical Manual

Version 3.2

www.radisys.com
World Headquarters
5445 NE Dawson Creek Drive • Hillsboro, OR
97124 USA
Phone: 503-615-1100 • Fax: 503-615-1121
Toll-Free: 800-950-0044

International Headquarters
Gebouw Flevopoort • Televisieweg 1A
NL-1322 AC • Almere, The Netherlands
Phone: 31 36 5365595 • Fax: 31 36 5365620

RadiSys Microware Communications Software Division, Inc.
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Revision A
April 2002

Copyright and publication information

This manual reflects version 3.2 of OS-9.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation 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. RadiSys Corporation 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 RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

April 2002

Contents

Chapter 1: System Overview

Chapter 2: The Kernel

Chapter 3: The OS-9 Input/Output System

Chapter 4: Interprocess Communication

Chapter 5: Subroutine Libraries and Trap Handlers

Chapter 6: OS-9 File System

Chapter 7: Resource Locking

Chapter 8: OS-9 System Calls

Appendix A: Example Code

Appendix B: OS-9 Error Codes

Index

1 System Overview

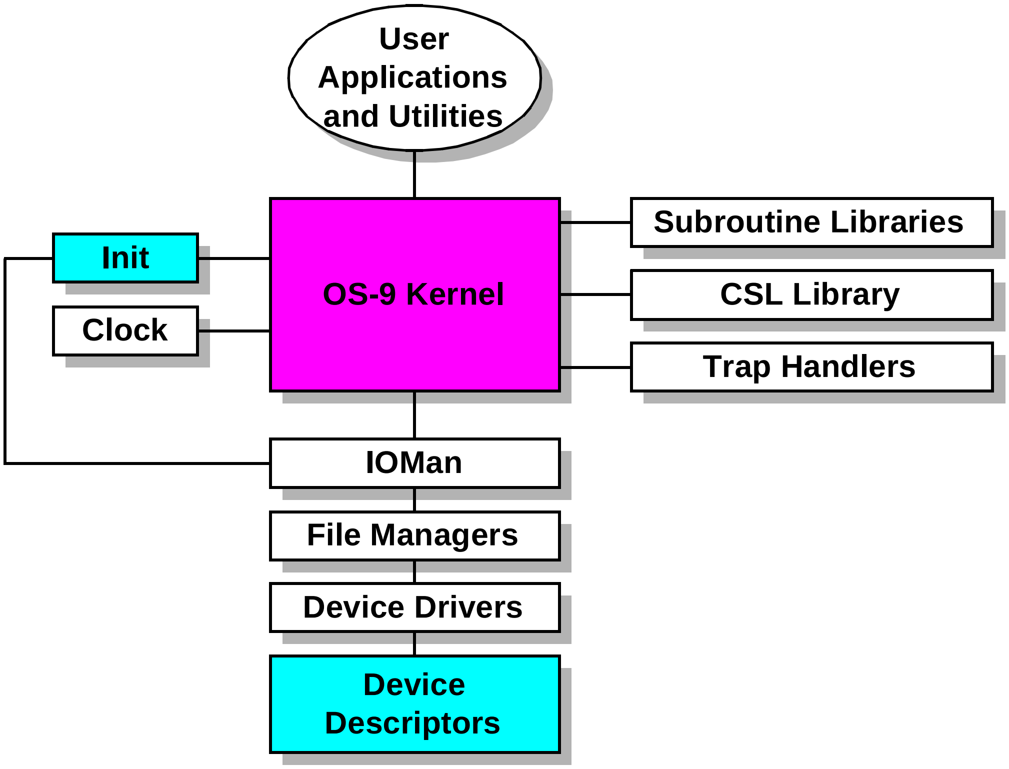
This chapter provides a general overview of OS-9 system modularity, I/O processing, memory modules, and program modules. It includes the following topics:

- System Modularity
- I/O Overview
- Memory Modules

SYSTEM MODULARITY

OS-9 has five levels of modularity. These are illustrated in Figure 1-1.

Figure 1-1. OS-9 Module Organization



Level 1 -- The Kernel, Clock, and Init Modules

The kernel provides basic system services, including process control and resource management. The clock module is a software handler for the specific real-time clock hardware. The kernel uses the Init module as an initialization table during system startup.

Level 2 — IOMAN

IOMAN coordinates the input/output (I/O) system by passing I/O requests to the appropriate file managers.

For specific information about IOMAN, file managers, device drivers, and device descriptors, refer to I/O Overview, *Chapter 2, The Kernel*, and the OS-9 Porting Guide.

Level 3 — 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 for OS-9.

Level 4 — 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 add customized drivers of your own design or purchase drivers from a hardware vendor.

Level 5 — Device Descriptors

Device descriptors are small tables that associate specific I/O ports with their logical name, device driver, and file manager. These modules also contain the physical address of the port and initialization data.

One important component not shown is the shell, which is the command interpreter. The shell is an application program, not part of the operating system, and is described in the *Using OS-9* manual.

For a list of the specific modules comprising OS-9 for your system, use the `ident` utility on the `sysboot` file.

Although all modules can 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

The OS-9 kernel does not directly process I/O requests. Instead, the kernel passes I/O requests to the I/O manager (IOMAN), and IOMAN passes requests to the appropriate file managers. Microware includes the following file managers in the OS-9 for Embedded Systems and Board Level Solution package:

Table 1-1. File Managers

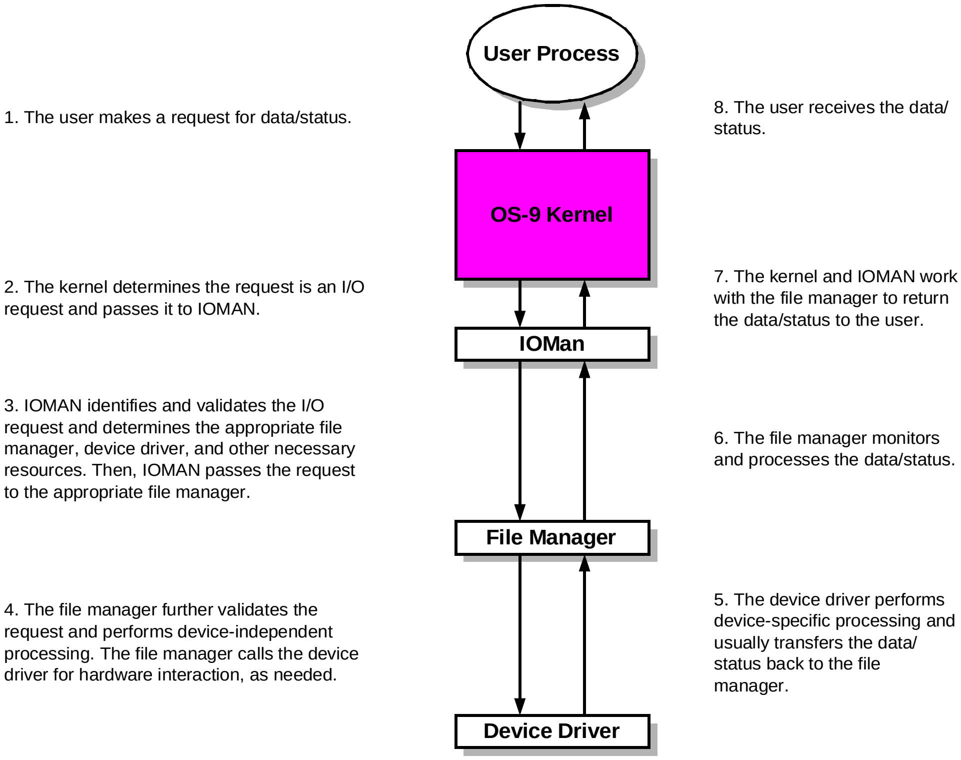
File Manager	Description
RBF	The Random Block File manager handles I/O for random-access, block-structured devices such as disks and hard drives.
SCF	The Sequential Character File manager handles I/O for sequential-access, character-structured devices such as terminals, printers, and modems.
SBF	The Sequential Block File manager handles I/O for sequential-access, block-structured devices.
PIPEMAN	The Pipe file Manager handles I/O for interprocess communications through memory buffers called pipes.
PCF	The PC file manager handles reading and writing to PC-DOS disks.

For more information about these file managers, refer to *Chapter 2, The Kernel*, or the *OS-9 Porting Guide***.**

Microware also supports additional communication file managers. Refer to the SoftStax and Lan Communications Pak manual sets for details.

Figure 1-2 illustrates how an OS-9 I/O request is processed.

Figure 1-2. Processing an OS-9 I/O Request


User Process
OS-9 Kernel
IOMan
File Manager
Device Driver
1. The user makes a request for data/status.
2. The kernel determines the request is an I/O request and passes it to IOMAN.
3. IOMAN identifies and validates the I/O request and determines the appropriate file manager, device driver, and other necessary resources. Then, IOMAN passes the request to the appropriate file manager.
4. The file manager further validates the request and performs device-independent processing. The file manager calls the device driver for hardware interaction, as needed.
5. The device driver performs device-specific processing and usually transfers the data/status back to the file manager.
6. The file manager monitors and processes the data/status.
7. The kernel and IOMAN work with the file manager to return the data/status to the user.
8. The user receives the data/status.

MEMORY MODULES

OS-9 is unique because it manages both the physical assignment of memory to programs and the logical contents of memory by using memory modules. A memory module is a logical, self-contained program, program segment, or collection of data.

OS-9 supports nine predefined module types and enables you to define your own module types. Each type of module has a different function. The predefined module types are defined in the `m_tylan` field of the module header definition.

Modules do not have to be complete programs or written in machine language. Modules simply have to be re-entrant, position independent, and conform to the basic module structure described in the next section.

OS-9 is based on a programming style called re-entrant code. That is, 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 effect each other, provided each process has an independent area for its variables.

Almost all OS-9 family software is re-entrant and uses memory efficiently. For example, a screen editor may require 26K of memory to load. If a request to run the editor is made while another user (process) is running it, OS-9 allows both processes to share the same copy, saving 26K of memory.

Data modules are an exception to the no-modification restriction. However, careful coordination is required for several processes to update a shared data module simultaneously.

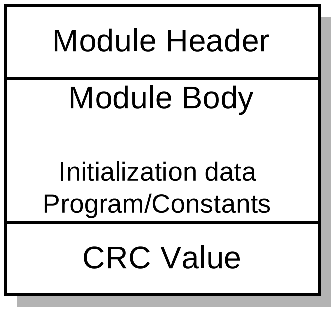
A position-independent module is in no way dependent on, or aware of where it is loaded in memory. This enables OS-9 to load the program wherever memory space is available. In many operating systems, the user must specify a load address to place the program in memory. OS-9 determines an appropriate load address only when the program is started.

OS-9 compilers and interpreters automatically generate position-independent code. In assembly language programming, however, you must insure position independence by avoiding absolute address modes. Alternatives to absolute addressing are described in the Assembler and Linker chapters of the ***Using Ultra C/C++*** manuals.

Basic Module Structure

Each module has three parts: a module header, a module body, and a CRC value as shown in Figure 1-3.

Figure 1-3. Basic Memory Module Format



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 name, size, type, language, memory requirements, and entry point. For specific information about the structure and individual fields of the module header, refer to the Module Header Definitions section in this chapter.

The module body contains initialization data, program instructions, and constant tables. The last three bytes of the module hold a CRC (cyclic redundancy check) value used to verify the module integrity when the module is loaded into memory. The linker creates the CRC at link time.

The CRC Value

A CRC (cyclic redundancy check) value is at the end of all modules, except data modules. The CRC, which is used to validate the entire module, is an error checking method used frequently in data communications and storage systems. The CRC is also a vital part of the ROM memory module search technique. It provides a high degree of confidence that programs in memory are intact before execution and is an effective backup for the error detection systems of disk drives and memory systems.

In OS-9, a 24-bit CRC value is computed over the entire module starting at the first byte of the module header and ending at the byte just before the CRC. OS-9 compilers and linkers automatically generate the module header and CRC values. If required, a user program can use the `F_CRC` system call to compute a CRC value over any specified data bytes. For a full description of how `F_CRC` computes a CRC value, refer to the description of the `F_CRC` call in *Chapter 8, OS-9 System Calls*.

In the case of data modules, the CRC value is not calculated when created. The CRC must be calculated and set on a data module before that module is loaded into memory.

OS-9 cannot recognize a module with an incorrect CRC value. For this reason, you must update the CRC value of a module modified in any way, or the module cannot be loaded from disk or located in ROM. Use the OS-9 `fixmod` utility to update the CRC of a modified module.

ROMed Memory Modules

When OS-9 starts after a system reset, the kernel searches for modules in ROM. The kernel detects the modules by looking for the module header sync code (for example, `0xf00d` for PowerPC processors). When this byte pattern is detected, the header parity is checked to verify a correct header. If this test succeeds, the module size is obtained from the header and a 24-bit CRC is computed over the entire module. If the CRC is valid, the module is entered into the module directory.

OS-9 links to all of its component modules found during the search. All ROMed modules present in the system at startup are automatically included in the system module directory. This enables you to create partially or completely ROM-based systems. Any non-system module found in ROM is also included. This enables user-supplied software to be located during the start-up process and entered into the module directory.

Module Header Definitions

The structure definition for a module header is listed here, followed by a description of each field.

mh_com

The module header structure is contained in the header file `module.h`.

Declaration

```
typedef struct mh_com {
     u_int16    m_sync,     /* sync bytes */
                m_sysrev;   /* system revision check value */
     u_int32    m_size;     /* module size */
     owner_id   m_owner;    /* group/user ID */
     u_int32    m_name;     /* offset to module name */
     u_int16    m_access,   /* access permissions */
                m_tylan,    /* module type and language */
                m_attrev,   /* module attributes and revision /*
                m_edit;     /* module edition number */
     u_int32    m_needs,    /* module hardware requirements flags */
                            /* (reserved) */
                m_share,    /* offset of shared data in statics */
                m_symbol,   /* offset to symbol table */
                m_exec,     /* offset to execution entry point */
                m_excpt,    /* offset to exception entry point*/
                m_data,     /* data storage requirement */
                m_stack,    /* stack size */
                m_idata,    /* offset to initialized data */
                m_idref,    /* offset to data reference lists */
                m_init,     /* offset to initialization routine*/
                m_term,     /* offset to termination routine */
                m_dbias,    /* data area pointer bias*/
                m_cbias;    /* code area pointer bias */
     u_int16    m_ident;    /* linkage locale identifier */
     char       m_spare[8]; /* reserved */
     u_int16    m_parity;   /* header parity */
} mh_com, *Mh_com;
```

Fields

`m_sync`
Constant bytes (for example, 0xf00d for the PowerPC) used to locate modules during the startup memory search. The value of `m_sync` is processor dependent.

`m_sysrev`
Identifies the format of a module.

`m_size`
Overall size of the module in bytes, including header and CRC.

`m_owner`
Group/user ID of the module’s owner.

`m_name` Contains the offset 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 (`0`) byte.

`m_access` Defines the permissible module access by its owner or by other users. The write permissions on memory modules only make sense for data modules. Module access permission values are located in the header file `module.h` and are defined as follows:

Name	Description
MP_OWNER_READ	$0001 = Read permission by owner
MP_OWNER_WRITE	$0002 = Write permission by owner
MP_OWNER_EXEC	$0004 = Execute permission by owner
MP_GROUP_READ	$0010 = Read permission by group
MP_GROUP_WRITE	$0020 = Write permission by group
MP_GROUP_EXEC	$0040 = Execute permission by group
MP_WORLD_READ	$0100 = Read permission by world
MP_WORLD_WRITE	$0200 = Write permission by world
MP_WORLD_EXEC	$0400 = Execute permission by world

All bits not defined in the preceding table are reserved.

`m_tylan`

Contains the module type (first byte) and language (second byte). The language codes indicate if the module is executable and which language the run-time system requires for execution, if any. Module type values and language codes are located in the header file `module.h` and are defined as follows:

Module Type	**Description**
MT_ANY	0 = Not used (wildcard value in system calls)
MT_PROGRAM	1 = Program module
MT_SUBROUT	2 = Subroutine module
MT_MULTI	3 = Multi-module (reserved for future use)
MT_DATA	4 = Data module
MT_CDBDATA	5 = Configuration Data Block data module 6-10 = Reserved for future use
MT_TRAPLIB	11 = User trap library
MT_SYSTEM	12 = System module
MT_FILEMAN	13 = File manager module
MT_DEVDRVR	14 = Physical device driver
MT_DEVDESC	15 = Device descriptor module 16-up = User definable

Language Code	**Description**
ML_ANY	0 = Unspecified language (wildcard in system calls)
ML_OBJECT	1 = Machine language
ML_ICODE	2 = Basic I-code (reserved for future use)

Language Code	Description
ML_PCODE	3 = Pascal P-code (reserved for future use)
ML_CCODE	4 = C I-code (reserved for future use)
ML_CBLCODE	5 = Cobol I-code (reserved for future use)
ML_FRTNCODE	6 = Fortran 7-15 = Reserved for future use 16-255 = User definable

Not all combinations of module type codes and languages are compatible.

`m_attrev`

Contains the module attributes (first byte) and revision (second byte). The attribute byte is defined in the header file `module.h` and as follows:

Bit	Description
7	The module is re-entrant (sharable by multiple tasks).
6	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.
5	The module is a system-state module.

If two modules with the same name and type are found in the memory search or are loaded into the current module directory, 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`
Indicates the software release level for maintenance. OS-9 does not use this field. Whenever a program is revised (even for a small change), increase this number. Internal documentation within the source program can be keyed to this system.

`m_needs`
Module hardware requirements flags (reserved for future use).

`m_share`
Offset to any shared data the module contains within its global data area. For example, this field is used by IOMAN to locate the main statics storage structure of file managers and device drivers.

`m_symbol`
Reserved.

`m_exec`
Offset to the program starting address, relative to the module starting address.

`m_excpt`
Relative address of a routine to execute if an uninitialized user trap is called.

`m_data`
Required size of the program data area (storage for program variables).

`m_stack`
Minimum required size of the program's stack area.

`m_idata`
Offset to an eight-byte value which precedes the initialized data area. The first four bytes contain an offset from the beginning of the program's memory to the beginning of the initialized data area, which contains values to copy to the program data area. The linker places all constant values declared in `vsects` here. The second four bytes contain the number of initialized data bytes to follow.

`m_idref`

Offset to a table of values to locate pointers in the data area. Initialized variables in the program’s data area may contain pointers to absolute addresses. Code pointers are adjusted by adding the absolute starting address of the object code area. Data pointers are adjusted by adding the absolute starting address of the data area.

`F_FORK` automatically calculates the effective address at execution time using the 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. The adjustment is made by combining the MS word with each LS word entry. This offset locates the pointer in the data area. The pointer is adjusted by adding 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, another MS word and LS word are given. This continues until an MS word of zero and an LS word of zero are found.

`m_init`

Offset to the trap handler initialization routine.

`m_term`

Reserved.

`m_dbias`

This field contains the bias value applied by the linker to the global data accesses in the module. Biasing global data accesses allows the compiler to generate efficient data accesses to a larger data space.

`m_cbias`

This field contains the bias value applied by the linker to the code symbols within the module. Biasing code references allows the compiler to generate efficient code references to a larger area of code.

`m_ident` Linkage site identifier. This field is not currently implemented.

`m_spare` Reserved.

`m_parity` One’s complement of the exclusive-OR of the previous header `words`. OS-9 uses this field to check module integrity.

2 The Kernel

This chapter outlines the primary functions of the kernel. It includes the following topics:

- Kernel Functions
- System Call Overview
- Kernel System Call Processing
- Memory Management
- OS-9 Memory Map
- Memory Fragmentation
- Colored Memory
- System Initialization
- Extension Modules
- Process Creation
- Process Scheduling

KERNEL FUNCTIONS

The nucleus of OS-9 is the kernel, which manages resources and controls processing. The kernel is a ROMable, compact, OS-9 module written in C language. The primary responsibility of the kernel is to process and coordinate system calls or service requests.

OS-9 has two general types of system calls. These include I/O calls, such as reads and writes, and system function calls.

System functions include those listed below:

- Memory management
- System initialization
- Process creation and scheduling
- Exception/interrupt processing

When a system call is made, the processor is changed to privileged state. The way this is done depends on which processor is being used. The kernel determines what type of system call you want to perform. The kernel directly executes the calls that perform system functions, but does not execute the I/O calls. Instead, the I/O calls are passed to IOMAN.

SYSTEM CALL OVERVIEW

For information about specific system calls, refer to *Chapter 8, OS-9 System Calls*.

User State and System State

There are two distinct OS-9 environments in which you can execute object code:

user state

User state is the normal program environment in which processes are executed. Generally, user-state processes do not deal directly with the specific hardware configuration of the system.

system state

System state is the environment in which OS-9 system calls and interrupt service routines are executed.

Functions executing in system state have several advantages over those running in user state:

- A system-state routine has access to the entire capabilities of the processor. For example, on memory protected systems, a system-state routine may access any memory in the system. It may mask interrupts, alter internal data structures, or take direct control of hardware interrupt vectors.
- System-state routines are never time sliced. Once a process has entered system state, no other process executes until the system-state process finishes or goes to sleep (F_SLEEP waiting for I/O). The only processing that may preempt a system-state routine is interrupt servicing.
- Some OS-9 system calls are only accessible from system state.

The characteristics of system state make it the only way to provide certain types of programming functions. For example, it is almost impossible to provide direct I/O to a physical device from user state. However, do not run all programs in system state.

- Signal handler routines cannot be called for system state processes. The process must dequeue them manually.
- In a multi-user environment, it is important to ensure each user receives a fair share of the CPU time. This is the basic function of time slicing.
- Memory protection prevents user-state routines from accidentally damaging data structures they do not own.
- A user-state process may be aborted. If a system-state routine loses control, the entire system usually crashes.
- It is far more difficult and dangerous to debug system-state routines 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 locate.

- User programs almost never have to be concerned with physical hardware; they are essentially isolated from it. This makes user-state programs easier to write and port.

⚠ If a system call returns with an exception error code, a system state exception has occured. If you are getting system state exceptions, there is a bug either in the OS-9 system code, a driver, or some user developed system code. Be advised that such exceptions can leave the system and the user program in an unknown and unstable state.

Installing System-State Routines

With direct access to all system hardware, any system-state routine has the ability to take over the entire machine. It is often a challenge to keep system-state routines from crashing or hanging up the system. increase system stability, the methods of creating routines that operate in system state are limited.

In OS-9, there are four ways to provide system-state routines:

1. Install an `OS9P2` 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. Typically, the `OS9P2` module is used to install new system service requests.

2. Use the I/O system as an entry into system state.

   File managers and device drivers are always executed in system state. In fact, the most obvious reason to write system-state routines is to provide support for new hardware devices. It is possible to write a dummy device driver and use the `I_GETSTAT` or `I_SETSTAT` routines to provide a gateway to the driver.

3. Write a trap handler module.

   For routines of limited use that are to be dynamically loaded and unlinked, this is perhaps the most convenient method. It is often practical to debug trap handler routines as user-state subroutines and then convert the finished routines to a trap handler module. OS-9 trap handlers always execute in system state.

4. Set the supervisor state bit in the attribute/revision word for the module.

   A program executes in system state if the supervisor state bit in the module attribute/revision word is set and if the module is owned by the super user.

KERNEL SYSTEM CALL PROCESSING

The kernel processes all OS-9 system calls (service requests). System call parameters are passed and returned in parameter blocks.

There are two general types of system calls:

- Non-I/O calls (calls performing system functions)
- I/O calls

System calls are identified by a function code passed in the service request parameter block. Every standard OS-9 system call has an associated symbolic name for the function code provided in the `funcs.h` C header file. The non-I/O call symbols begin with `F_` and the I/O calls begin with `I_`. For example, the system call to link a module is called `F_LINK`.

Non-I/O Calls

There are two types of non-I/O system calls:

- **User-State System Calls** — Perform memory management, multitasking, and other functions for user programs. These are mainly processed by the kernel.
- **System-State System Calls** — Can only be used by system software in system state and usually operate on internal OS-9 data structures. To preserve the modularity of OS-9, these requests are system calls rather than subroutines. User-state programs cannot access these calls, but system modules such as device drivers can use these calls.

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

Memory requested in system-state is not recorded in the process descriptor memory list. The requesting process must ensure the memory is returned to the system before the process terminates.

I/O Calls

When the kernel receives an I/O request, it immediately passes the request to IOMAN. IOMAN passes the request to the appropriate file manager and device driver for processing.

Any I/O system call may be used 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 user-state process has a path table used to convert its local path numbers to system path numbers. The system itself has a global path number table used to convert system path numbers into actual addresses of path descriptors. System-state I/O system calls must be made using system path numbers.

For example, a system-state OS-9 `I_WRITE` system call prints an error message on the caller's standard error path. To do this, a system-state process may not perform output on path number two. Instead, it must use the `I_TRANPN` system call to translate the user path number to its associated system path number.

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 is not recorded in the process path table or anywhere else by OS-9; the system-state routine that opens an I/O path must ensure the path is eventually closed. This is true even if the underlying process is abnormally terminated.

MEMORY MANAGEMENT

If any object (such as a program and constant table) is to be loaded in memory, it must use the standard OS-9 memory module format described in *Chapter 1, System Overview*. This enables 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.

After OS-9 has been booted, a single module directory exists containing all of the boot modules. You may create additional module directories and subdirectories at your discretion. Each module directory has independent access permissions. By using multiple module directories, modules with the same name can be loaded in memory and executed without conflict.

This can be extremely useful in the continuing development of existing software. When a module is loaded in memory, it is added to the process current module directory.

When a process creates a new process, the OS-9 kernel does the following:

1. Searches the current module directory for the target module.
2. If this search fails, the kernel searches the process' alternate module directory, initially specified in your login file.
3. If this search fails, the kernel attempts to load the module into the current module directory.

Each module 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 link count of the module is incremented by one. When a process unlinks from a module, the link count is decremented by one. When a module's link count becomes zero, its memory is deallocated and the module 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 byte (first byte of the `m_attrev` field) is set. If several modules are merged together and loaded, you must unlink all of those modules before any are removed from the module directory.

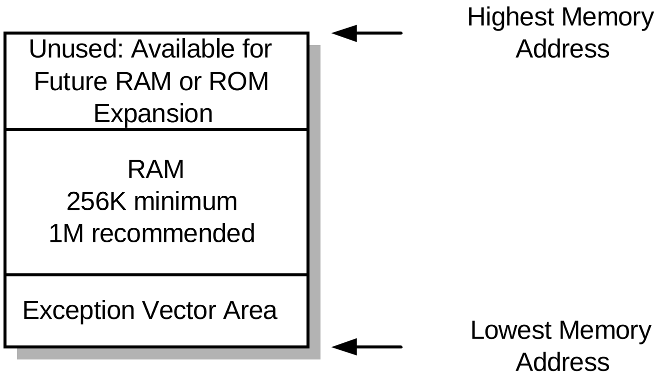
Refer to Chapter 5 of *Using OS-9* for more information on module directories.

OS-9 MEMORY MAP

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

A map of an example OS-9 memory space is shown in Figure 2-1. The sections shown are not required to be at specific addresses. Microware recommends you keep each section in contiguous reserved blocks arranged in an order that facilitates future expansion. It is always advantageous for RAM to be physically contiguous whenever possible.

Figure 2-1. Example OS-9 Memory Map



System Memory Allocation

During the OS-9 start-up sequence, an automatic search function in the kernel and the boot ROM locates 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.

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

Operating System Object Code

On disk-based systems, operating system component modules (such as the kernel, I/O managers, and device drivers) 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. Alternately, you can place OS-9 in ROM for non-disk systems.

System Global Memory

The OS-9 kernel allocates a section of RAM memory for internal use. It contains the following items:

- An exception jump table
- The debugger/boot variables
- A system global area

Variables in the system global area are symbolically defined in the `sysglob.h` library and the variable names begin with `d_`.

User programs should never directly access system global variables. System calls are provided to allow user programs to read the information in this area.

System Dynamic Memory

OS-9 maintains dynamic-sized data structures (such as I/O buffers, path descriptors, and process descriptors) that are allocated from the general RAM area when needed. The system modules allocate and maintain these structures. For example, IOMAN allocates memory for path descriptors and maintains them. The system global memory area contains the pointers to these system data structures.

User Memory

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

- Modules are loaded in RAM.
- New processes are created.
- Processes request additional RAM.
- OS-9 requires more I/O buffers.
- OS-9 internal data structures must be expanded.

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

The total memory required for user memory depends largely on the application software to be run.

MEMORY FRAGMENTATION

Once a program is loaded, it remains 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 dynamically relocated afterwards. This characteristic can lead to a troublesome phenomenon called memory fragmentation.

When programs are loaded, they are assigned the first sufficiently large block of memory at the highest address possible in the address space. (If a colored memory request is made, this may not be true. Refer to the following section for more information on colored memory.)

If a number of program modules are loaded, and subsequently one or more non-contiguous modules are unlinked, several fragments of free memory space will exist. The total free memory space may be quite large. Because it is scattered, however, not enough space exists in a single block to load a particular program module.

You can avoid memory fragmentation by loading modules at system startup. This places the modules in contiguous memory space. You can also initialize each standard device when the system is booted. This enables the devices to allocate memory from higher RAM than would be available if the devices were initialized later.

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. The `mfree` utility can 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, part of a RAM can store video images and another part can be battery-backed. The kernel allows areas of RAM like these to be isolated and accessed specifically. You can request specific memory types or colors when you allocate memory buffers, create modules in memory, or load modules into memory. If a specific type of memory is not available, the kernel returns error `#237, EOS_NORAM`.

Colored memory lists are not essential on systems whose RAM consists of one homogeneous type, although they can improve system performance on some systems and allow greater flexibility in configuring memory search areas.

Colored Memory Definition List

The kernel must have a description of the CPU address space in order to use the colored memory routines. This is accomplished by including a colored memory definition list in `default.des` file. The list describes the characteristics of each memory region. The kernel searches each region in the list for RAM during system startup.

The following information describes a memory area to the kernel:

- Memory color (type)
- Memory priority
- Memory access permissions
- Local bus address
- Block size to be used by the kernel cold start routine to search the area for RAM or ROM
- External bus translation address (for DMA and dual-ported RAM)
- Optional name

The memory list (`memlist`) 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.

Each line in the memory list must contain all the parameters in the following order: type, priority, attributes, blksiz, addr begin, addr end, name, and DMA-offset.

The colored memory list must end on an even address. Descriptions of the `memlist` fields are included below:

Table 2-1. memlist Fields

Parameter	Size	Definition
Memory Type	word	Type of memory. Two memory types are currently defined in `memory.h`: `MEM_SYS` `0x01` System RAM memory `MEM_SHARED` `0x8000` Shared memory (0x8000 - 0xffff)
Priority	word	High priority RAM is allocated first (255 - 0). 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 Fields (Continued)

Parameter	Size	Definition

Access Permissions	word	Memory type access bit definitions:
		`bit 0` `B_USERRAM`
		Indicates memory allocatable by user processes.
		NOTE: This bit is ignored if the `B_ROM` bit is also set.
		`bit 1` `B_PARITY`
		Indicates parity memory; the kernel initializes it during start-up.
		`bit 2` `B_ROM`
		Indicates ROM; the kernel searches this for modules during start-up.
		`bit 3` `B_NVRAM`
		Non-volatile RAM; the kernel searches this for modules during start-up.
		`bit 4` `B_SHARED`
		Shared memory; reserved for future use.
		NOTE: Only `B_USERRAM` memory may be initialized.

Search Block Size	word	The kernel checks every search block size to see if RAM/ROM exists.
Low Memory Limit	long	Beginning address of the block as referenced by the CPU.
High Memory Limit	long	End address of the block as referenced by the CPU.
Description String Offset	long	This 32-bit offset of a user-defined string describes the type of memory block.
Address Translation Adjustment	long	External bus address of the beginning of the block. If zero, this field does not apply. Refer to `_os_trans()` for more information.

Refer to your ***OS-9 Device Descriptor and Configuration Module Reference*** for more information on creating a memory list in the init modules.

The complete memory list structure definitions are located in the `alloc.h` file and are listed here:

```
/* initialization table (in memdefs module data area) */
typedef struct mem_table {
     u_int16
          type,       /* memory type code */
          prior,      /* memory allocation priority */
          access,     /* access permissions */
          blksiz;     /* search block size */
     u_char
          *lolim,     /* beginning absolute address for this type */
          *hilim;     /* ending absolute address +1 for this type */
     u_int32
          descr;      /* optional description string offset */
     u_int32
          dma_addr,   /* address translation address for dma's, etc.*/
          rsvd2[2];   /* reserved, must be zero */
} *Mem_tbl, mem_tbl;

/* access bit definitions */
#define B_USERRAM      (0x01)     /* memory allocatable by user procs */
#define B_PARITY       (0x02)     /* parity memory; must be initialized */
#define B_ROM          (0x04)     /* read-only memory; searched for modules */
#define B_NVRAM        (0x08)     /* non-volatile RAM; searched for modules */
#define B_SHARED       (0x10)     /* shared memory (Reserved for future use.)*/
```

SSM and Cache

The SSM module provides user-state security. The cache module is used to enforce cacheing policy on ranges of memory. The following sections describe these modules and how they are used with OS-9 and its supported processors.

Cache List

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 you to configure the system for optimal performance and/or system functionality. In many cases, 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 “writeback” for user-state accesses. This default mode can be over-ridden for specific regions by creating cache list entries in the `Init` module. The cache list is used to describe the cacheing policy enforced by the cache module.

The cache list entries must end with a longword of 0xffffffff (-1). The following table 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 cache mode is usually controlled by SSM, rather than the cache module. Cache mode information is located in the following location:

`MWOS/OS9000/SRC/DEFS/cache.h`

SSM and Cache for PowerPC

This section refers to the 603, 604, 750, 8240, and 8260 PowerPC processors. It explains how the SSM and cache modules work in order to provide system security and caching policy.

Cacheing policies consist of the following types of cache behaviors:

- Writeback or “copyback” cache is the default cache mode of a memory region. It is the fastest type of cache, and the most problematic. With copyback cache, values written to memory are first written to cache--not to main memory. During normal operation of the processor, data values held in cache are written to main memory. Under normal circumstances this is not a problem; the cache module provides services in which a programmer can force the write of data values in cache to main memory. This ensures that the data values in the cache are coherent with the data values in main memory. Typically, this is important in multi-processor environments, especially DMA devices. Processors and DMA devices can only read main memory, and not the caches of another processor.

- Write through cache is the fastest type of cache, after Copyback. Writes to this cache are written to the cache and to the memory immediately. Reads are still cached.
- The last type of cache is no cache at all. In other words, read and writes bypass the cache and go directly into memory.

Using the cachelist in the init module, all cacheing policies can be specified on memory ranges. The 600 series PowerPC processor has two mechanisms in order to enforce cache policy: the first is the memory segment model; the second is the block address translation (BAT) mechanism.

The memory segment model is used for user-state memory protection and security, as well as cache policy. The SSM module controls this mechanism. The minimum memory range to control is 4k bytes.

The cache module controls the BAT mechanism. In addition, if the SSM module is not included within the system, the BAT mechanism will be used for system-state and user-state cache policies.

The BAT mechanism is simpler than the memory segment module; however, its options are more limited. This limitation is reflected directly onto the cachelist in the init module. Minimum memory range for a BAT is 128k bytes; regions are in the multiple of power of two from through 256M bytes. Actual BAT memory sizes are as follows: 128k, 256k, 512k 1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M, and 256M. Each of these memory ranges must aligned according to their size, therefore, a 128k memory range must begin on a 128k memory address alignment, and a 32M memory range must begin on a 32M memory address alignment. There are four BAT registers on the processor; each defines only one memory range.

The cachelist for the PowerPC family of processors contains the following structure:

```
init cachelist[0] {        /*  Initialize cache list element
*/
    blk_beg = 0x00000000;  /*  beginning address of memory region
*/
    blk_end = 0x02000000;  /*  ending address +1
*/
    mode = 0;              /*  cache mode of region */
};
```

The cachelist consists of two parts: the system-state list and the user-state list. The system-state list is the cacheing policy of memory accessed from system-state, and the user-state list is the cacheing policy of memory accessed from user-state. The user-state list follows the system-state list in the array. (Each list is terminated with an entry that has the beginning address of 0xffffffff.)

Where the mode is concerned, a 0 indicates copyback cache. Macros have been defined to enforce other modes. IO_CACHE_MODE is used no cache; CM_DWTHRU is used to write through mode.

Guidelines

In order to define a cachelist, the following guidelines should be followed:

- Minimum segment size should be 128k bytes.
- Minimum segment start address alignment should be 128k bytes.
- A segment start alignment address size must be less than or equal to the segment size itself. The 256M segment size is the exception. Multiple 256M segments can be defined as one region.
- The segment for address 0 must have sufficient memory to bring up the kernel. In other words, it must have enough memory for the kernel's globals, IRQ stack, and bootfile--if it is loaded into RAM.
- The memory definitions for different segments must not overlap.
- The SSM module must be placed before the cache module in the extension list. In other words, the SSM module must run before the cache module can run if both are in the system. The SSM or the cache module can run stand-alone as well.
- The user-state cachelist normally inherits system-state list entries. Only specific user-state cacheing differences need to be listed in the user-state cache list. For example, video memory made accessible to user-state needs to be marked as IO_CACHE_MODE, or with a data cache write-through, with the user-state driver forcing stores or flushes.

Example

The following is a typical cachelist for a PowerPC system:

```
/* System-state Dram */
init cachelist[0] {          /*  Initialize cache list element          */
    blk_beg = 0x00000000;    /*  beginning address of memory region     */
    blk_end = 0x10000000;    /*  ending address +1                      */
    mode = 0;                /*  cache mode of region                   */
};

/* The rest of memory is considered non-cached. */
init cachelist[1] {          /*  Initialize cache list element          */
    blk_beg = 0x10000000;    /*  beginning address of memory region     */
    blk_end = 0x00000000;    /*  ending address +1                      */
    mode = IO_CACHE_MODE;    /*  cache mode of region                   */
};

/* End of system-state cache list */
init cachelist[2] {          /*  Initialize cache list element          */
    blk_beg = 0xffffffff;    /*  beginning address of memory region     */
    blk_end = 0x00000000;    /*  ending address +1                      */
    mode = 0;                /*  cache mode of region                   */
};
/* User-state list DRAM */
init cachelist[3] {          /*  Initialize cache list element          */
    blk_beg = 0x00000000;    /*  beginning address of memory region     */
    blk_end = 0x02000000;    /*  ending address +1                      */
    mode = 0;                /*  cache mode of region                   */
};

/* End of user list */
init cachelist[4] {          /*  Initialize cache list element          */
    blk_beg = 0xffffffff;    /*  beginning address of memory region     */
    blk_end = 0x00000000;    /*  ending address +1                      */
    mode = 0;                /*  cache mode of region                   */
};
```

With the above cachelist, there is 256M bytes of copyback cached dram; the rest of the system is considered non-cached.

Setup for Allocatable NON-CACHED Memory

By matching the memlist entries in the init module to the cachelist entries, a region of non-cached memory can be defined. In this way, anything in the system can use the non-cached memory as a resource.

The following entries define a region of 512k bytes of non-cached memory out of 32M of total memory. You should define such regions at the end of physical dram.

```
/* Memlist entries */
init memlist[0] {           /*  Initialize memory list element              */
    type = MEM_SYS;         /*  memory type code                            */
    prior = 0xFF;           /*  memory allocation priority                  */
    access = B_USERRAM;     /*  access permissions                          */
    blksiz = 0x800;         /*  search block size                           */
    lolim = 0x00080000;     /*  beginning absolute address                  */
    hilim = 0x01F80000;     /*  ending absolute address + 1                 */
};

init memlist[1] {           /*  Initialize memory list element              */
    type = MEM_NOCACHE;     /*  memory type code                            */
    prior = 0x0;            /*  memory allocation priority                  */
    access = B_USERRAM;     /*  access permissions                          */
    blksiz = 0x800;         /*  search block size                           */
    lolim = 0x01F80000;     /*  beginning absolute address                  */
    hilim = 0x02000000;     /*  ending absolute address + 1                 */
};

/* End of memlist */
init memlist[2] {           /*  Initialize memory list element              */
    type = 0;               /*  memory type code                            */
    prior = 0x0;            /*  memory allocation priority                  */
    access = 0;             /*  access permissions                          */
    blksiz = 0x0;           /*  search block size                           */
    lolim = 0x0;            /*  beginning absolute address                  */
    hilim = 0x0;            /*  ending absolute address + 1                 */
};
/*  Cachelist entires */
/* DRAM */
init cachelist[0] {         /*  Initialize cache list element           */
    blk_beg = 0x00000000;   /*  beginning address of memory region      */
    blk_end = 0x01F80000;   /*  ending address +1                       */
    mode = 0;               /*  cache mode of region                    */
};

init cachelist[1] {         /*  Initialize cache list element           */
    blk_beg = 0x01F80000;   /*  beginning address of memory region      */
    blk_end = 0x02000000;   /*  ending address +1                       */
    mode = IO_CACHE_MODE;   /*  cache mode of region                    */
};

/* The rest of memory is considered non-cached. */
init cachelist[2] {         /*  Initialize cache list element           */
    blk_beg = 0x10000000;   /*  beginning address of memory region      */
    blk_end = 0x00000000;   /*  ending address +1                       */
    mode = IO_CACHE_MODE;   /*  cache mode of region                    */
};

/* End of system-state cache list */
init cachelist[3] {         /*  Initialize cache list element           */
    blk_beg = 0xffffffff;   /*  beginning address of memory region      */
    blk_end = 0x00000000;   /*  ending address +1                       */
    mode = 0;               /*  cache mode of region                    */
};
```

```
/* DRAM */
init cachelist[4] {         /*  Initialize cache list element           */
    blk_beg = 0x00000000;   /*  beginning address of memory region      */
    blk_end = 0x01f80000;   /*  ending address +1                       */
    mode = 0;               /*  cache mode of region                    */
};

init cachelist[5] {         /*  Initialize cache list element           */
    blk_beg = 0x01f80000;   /*  beginning address of memory region      */
    blk_end = 0x02000000;   /*  ending address +1                       */
    mode = IO_CACHE_MODE;   /*  cache mode of region                    */
};
init cachelist[6] {         /*  Initialize cache list element           */
    blk_beg = 0xffffffff;   /*  beginning address of memory region      */
    blk_end = 0;            /*  ending address +1                       */
    mode = 0;               /*  cache mode of region                    */
};
```

The MEM_NOCACHE type of memory is defined as memory that is not cached. Once defined in this manner, system drivers and other code can use the non-cached memory.

The memlist entry for MEM_NOCACHE has a priority of 0.

The cache module is responsible for splitting up the cache list into a BAT array list. This list is sorted, and the cache module has an exception handler that will replace BAT registers in the processor to allow access, and cache policy into the particular region of memory.

BAT register 0 contains the entry for address 0 and is never replaced. BAT registers 1, 2, and 3 are all capable of being replaced via “round robin”. This BAT array list contains all of the individual entries defined by the cachelist. Each cachelist entry may get split up into several BAT array elements as needed. For example, in the cachelist[2] entry above, the entire 3.75G byte area splits into 15 BAT array elements, each one 256M bytes in size. Because there is no 31.5M byte BAT entry size in cachelist[0], the cachelist[0] entry splits into six BAT array elements. The breakdown is as follows:

1 16M segment

1 8M segment

1 4M segment

1 2M segment

1 1M segment

1 512K segment

The cachelist[1] entry takes up the other 512K that makes up the 32M of physical memory contained by the computer.

Performance Considerations

The above BAT array scheme can have a negative impact on performance. Although the first segment is locked into BAT register 0, the other five segments in cachelist[0], the segment in cachelist[1], and other segments in cachelist[2] for I/O regions contend for the other three BAT registers. However, there are some procedures for handling this.

- If you do not need non-cached memory, do not use it. Defining a small non-cached memory region results more in the necessary replacement of some segments than a larger non-cached memory region.
- Define differing priorities of memory. If segment 1 is defined alone as a high priority memory in the memlist, and the other segments grouped into a lower priority memory in the memlist, all of the beginning kernel setup and processes will run from the first segment. The other memory priorities will only be allocated when this segment has run out of memory.

Colored Memory in Homogenous Memory Systems

As previously mentioned, colored memory definitions are not essential for systems whose memory is homogenous. However, these types of systems can benefit from this feature of the kernel in terms of system performance and ease of memory list reconfiguration.

System Performance

In a homogeneous memory system, the kernel allocates memory from the top of available RAM when requests are made by `F_SRQMEM` (loading modules). If the system has RAM on-board the CPU and off-board in external memory boards with higher addresses, the modules tend to be loaded in the off-board RAM. On-board RAM is not used for a `F_SRQMEM` call until the off-board memory cannot accommodate the request.

Due to bus access arbitration, programs running in off-board memory execute more slowly than if they were executing in on-board memory. Also, external bus activity is increased. This may impact the performance of other bus masters in the system.

The colored memory lists can reverse this tendency in the kernel, so a CPU can 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. To do this, make the priority of the on-board memory higher than the off-board memory.

Reconfiguring Memory Areas

In a homogeneous memory system, the memory search areas are defined in the ROM memory list. Changes to these areas previously required new ROMs be made from source code (usually impossible for end users) or from a patched version of the original ROMs (usually difficult for end users).

The colored memory lists somewhat alleviate this situation by configuring the search areas as follows:

- The ROM memory list describes only the on-board memory.
- The colored memory lists in `default.des` define any external bus memory search areas in the Init module only.

Using colored memory in this situation enables the end user to easily reconfigure the external bus search areas by adjusting the lists in `default.des` and making a new Init module. The ROM does not require patching.

SYSTEM INITIALIZATION

After a hardware reset, the kernel (located in ROM or loaded from disk, depending on your system configuration) is executed by the bootstrap ROM. The kernel initializes the system; this includes locating ROM modules and running the system start-up task.

Init: The Configuration Module

The `init` module:

- Is non-executable module of type `MT_SYSTEM`
- Contains a table of system start-up parameters
- Specifies the initial table sizes and system device names during startup
- Is always available to determine system limits
- Is required to be in memory when the system is booting and usually resides in the `sysboot` file or in ROM

- Begins with a standard module header

  The `m_exec` offset in the module header is a pointer to the system constant table. The fields of this table are defined in the `init.h` header file.

  Refer to the *OS-9 Device Descriptor and Configuration Module Reference* for a listing of the init module fields.

EXTENSION MODULES

To enhance OS-9 capabilities, you can execute additional modules at boot time. These extension modules 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.

To include an extension module in the system, you can either program the module into system memory or use the `p2init` utility to add it to a running system.

Refer to the *Utilities Reference* for information about `p2init`. Refer to the *OS-9 Device Descriptor and Configuration Module Reference* for procedures to change the `init` modules and your board guide for instructions on how to build a new boot file containing the desired extension modules.

When an extension module is called for initialization during coldstart, the module's entry point is executed with its global static storage (if any) pre-initialized and set. The extension module is passed a pointer to the kernel's global static storage as defined in the header file `sysglob.h`.

PROCESS CREATION

All OS-9 programs are run as processes or tasks. New processes are created by the `F_FORK` system call. The most important parameter passed in the fork system call is the name of the primary module that the new process is to execute initially.

The following list outlines the creation process:

1. Locate or Load the Program.

   OS-9 searches for the module in memory by means of the module directory. If OS-9 cannot locate 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 and an I/O Descriptor.

   After the primary module has been located, a data structure called a process descriptor is assigned to the new process. The process descriptor is a table containing information about the process such as its state, memory allocation, and priority. The I/O descriptor contains information about the process I/O such as the I/O paths and counts of bytes read and written. The process descriptor and I/O descriptor are automatically initialized and maintained. Processes do not need to be aware of the existence or contents of process descriptors or I/O descriptors.

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. Process memory areas are discussed in the following section.

4. Initialize the Process.

   The new process' registers are set to the proper addresses in the data area and object code module. If the program uses initialized variables and/or pointers, they are copied from the object code area to the proper addresses in the data area.

If any of these steps cannot be performed, creation of the new process is aborted and the process that originated the **fork** is notified of the error. If all the steps are completed, the new process is added to the active process queue for execution scheduling.

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

Processes terminate when they execute an F_EXIT system service request or when they receive fatal signals or errors. Terminating the process performs the following functions:

- Closes any open paths
- Deallocates the process' memory
- Unlinks its primary module
- Unlinks any subroutine libraries or trap handlers the process may have used

Process Memory Areas

All processes are divided into two logically separate memory areas:

- code
- data

This division provides the modular software capabilities for OS-9.

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 automatic OS-9 functionality results in more 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 to be read-only. It may link to and execute code in other modules.

The process data area is a separate memory space where all of the program 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 to access the data area. You can read and write to this area.

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

Process States

A process can be in one of five states:

State	Description
Active	The process is active and ready for execution. Active processes are given time for execution according to their relative priority with respect to all other active processes. The scheduler uses a method that compares the ages of all active processes in the queue. All active processes receive some CPU time, even if they have a very low relative priority.
Event	The process is inactive until the associated event occurs. The event state is entered when a process executes an `F_EVENT` service request when the specified event condition is not satisfied. The process remains inactive until another process or interrupt service routine performs an `F_EVENT` system call that satisfies the waiting process's condition.
Sleeping	The process is inactive for a specific period of time or until a signal is received. The sleep state is entered when a process executes an `F_SLEEP` service request. `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 completing I/O. Zero ticks specifies an infinite period of time. A process waiting on an event waits in a queue associated with the specific event, but behaves as though it was in the sleep queue.

State	Description
Suspended	The process is inactive, unknown to the system, and not a member of any queue. The suspended state is entered when a process or system module does an F_SSPD call on a given process. The process can be reactivated with an F_APROC call.
Waiting	The process is inactive until a child process terminates or until a signal is received. When a process executes an F_WAIT system service request, it enters the wait state. The process remains inactive until one of its descendant processes terminates or until it receives a signal.

A separate queue (linked list of process descriptors) exists for each process state, except the suspended state. State changes are accomplished by moving a process descriptor from its current queue to another queue.

PROCESS SCHEDULING

OS-9 is a multitasking operating system. This means two or more independent programs, called processes, or tasks, can execute simultaneously. Each second of CPU time is shared by several processes. Although the processes appear to run continuously, the CPU only executes one instruction at a time. The OS-9 kernel determines which process to run and for how long, based on the priorities of the active processes.

The action of switching from the execution of one process to another is called task switching. Task switching does not effect program execution.

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

The longest amount of time a process controls the CPU before the kernel re-evaluates the active process queue is called a slice or time slice per slice at run-time, adjust the system global variable d_tslice.

You can also change the number of ticks per slice prior to booting the system by modifying m_slice in the init modules.

Refer to the *OS-9 Device Descriptor and Configuration Module Reference* for information to modify this field.

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' initial priority. The oldest process is at the head of the queue. The OS-9 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 assigned priority and the ages of all other processes are incremented. Ages are never incremented beyond 0xffff.

After the time slice of the currently executing process, the kernel executes the process with the highest age.

Preemptive Task Switching

During critical real-time applications, fast interrupt response time is sometimes necessary. 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 in 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 and are accessible by users with a group ID of zero (super users) through the F_SETSYS system call.

If the priority or age of a process is less than d_minpty, the process is not considered for execution and is not aged. Usually, this variable is not used and is set to zero.

If the minimum system priority is set above the priority of all running tasks, the system completely shuts down. It can only be recovered by a reset. This makes it crucial to restore d_minpty to a normal level when the critical task(s) finishes.

`d_maxage` is the maximum age to which processes can be incremented. When `d_maxage` is activated, tasks are divided into high priority tasks and low priority tasks.

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

3 The 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
- IOMAN
- Device Descriptor Modules
- Path Descriptors
- Access Modes and Permissions
- File Managers
- Device Driver Modules

THE OS-9 UNIFIED INPUT/OUTPUT SYSTEM

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

The I/O subsystem consists of three modules processing I/O service requests at different levels:

- The I/O Manager
- The File Manager
- The Device Driver

A fourth module, the device descriptor, contains the information used to assemble the different components of an I/O subsystem. The file manager, device driver, and device descriptor modules are standard memory modules you can install and remove dynamically while the system is running.

The I/O Manager

IOMAN manages the following four tasks:

- Supervises the OS-9 I/O system
- Establishes the connections between itself, the file manager, and the device driver
- Manages various data structures
- Ensures the appropriate file manager and device driver modules process a particular I/O request

The File Manager

A file manager performs the processing for a particular class of devices such as disks or terminals. For example, the Random Block File Manager (RBF) maintains directory structures on disks and the Sequential Character File manager (SCF) edits the data stream it receives from terminals.

The Device Driver

A device driver has the following three primary tasks:

- Enables OS-9 to be device independent
- Operates on the actual hardware device, sending data to and from the device on behalf of the file manager
- Isolates the file manager from actual hardware dependencies such as control register organization and data transfer modes

IOMAN

When the kernel receives an I/O request, it immediately passes the request to IOMAN. IOMAN provides the first level of service for 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 on behalf of file managers and device drivers.

Many controllers, such as SCSI interfaces and DUARTs (Dual Asynchronous Receiver-Transmitters), actually operate multiple devices. IOMAN allocates and initializes an additional static storage for each device, called logical unit static storage, to assist file managers and drivers with managing these interfaces.

IOMAN maintains two important internal data structures:

- The device list
- The path table

These tables reflect two other structures respectively:

- The device descriptor
- The path descriptor

When an `I_ATTACH` system call is first performed on a new device descriptor, IOMAN creates a new entry in the device list. Each entry in the device list contains information about each element required to perform I/O on a device.

A device list entry also contains pointers to the various static storages and other data elements in use on the device. The structure definition of a device list entry is defined in the header file `io.h`.

When a path is opened, IOMAN links 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. IOMAN saves the information in the device entry list of the device descriptor, so subsequent system calls can be routed to these modules.

Paths are used to 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.

DEVICE DESCRIPTOR MODULES

A device descriptor module is a small, non-executable module providing information that associates a specific I/O device with the following:

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

Device drivers and file managers operate on general classes of devices, not specific I/O ports. A device descriptor tailors its functions to a specific I/O port.

The name of the device descriptor 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 (`MT_DEVDESC`).

One device descriptor must exist for each I/O device in the system. However, one device can also have several device descriptors with different initialization constants.

The device descriptor contains a constant table and logical unit static storage initialization information. IOMAN initializes logical unit static storage with the `F_INITDATA` system call, similar to how other processing elements in the system initialize their static storage areas. IOMAN does not restrict the definition or use of logical unit static storage.

A constant table containing information provided by a device descriptor is located at the entry point offset of the device descriptor. IOMAN requires the first part to be common to all device descriptors. File managers and device drivers may add information they require after the common part. The format of the common part is shown here and defined in the header file `io.h`. Data defined by specific file managers is provided in the *OS-9 Device Descriptor and Configuration Module Reference*.

dd_com

Declaration

```
/* Device descriptor data definitions */
typedef struct {
     void      *dd_port;      /* device port address */
     u_int16   dd_lu_num,     /* logical unit number */
               dd_pd_size,    /* path descriptor size */
               dd_type,       /* device type */
               dd_mode;       /* device mode capabilities */
     u_int32   dd_fmgr,       /* file manager name offset */
               dd_drvr;       /* device driver name offset */
     u_int16   dd_class,      /* sequential or random */
               dd_dscres;     /* (reserved) */
} *Dd_com, dd_com;
```

Fields

`dd_port` Absolute physical address of the hardware controller.

`dd_lu_num` Distinguishes the different devices driven from a unique controller. Each unique number represents a different logical unit static storage area.

`dd_pd_size` Size of the path descriptor. Path descriptors vary in size. IOMAN uses this value when it allocates a path descriptor.

`dd_type` Identifies the I/O type of the device. The following values are defined in the header file `io.h`:

Defined Name	Value	Description
DT_SCF	0	Sequential Character File Type
DT_RBF	1	Random Block File Type
DT_PIPE	2	Pipe File Type
DT_SBF	3	Sequential Block File Type
DT_NFM	4	Network File Type

Defined Name	Value	Description
DT_CDFM	5	Compact Disc File Type
DT_UCM	6	User Communication Manager
DT_SOCK	7	Socket Communication Manager
DT_PTTY	8	Pseudo-Keyboard Manager
DT_GFM	9	Graphics File Manager
DT_PCF	10	PC-DOS File Manager
DT_NRF	11	Non-volatile RAM File Manager
DT_ISDN	12	ISDN File Manager
DT_MPFM	13	MPFM File Manager
DT_RTNFM	14	Real-Time Network File Manager
DT_SPF	15	Stacked Protocol File Manager
DT_INET	16	Inet File Manager
DT_MFM	17	Multi-media File Manager
DT_DVM	18	Generic Device File Manager
DT_NULL	19	Null File Manager
DT_DVDFM	20	DVD File Manager
DT_MODFM	21	Module Directory File Manager

DT-codes up to 127 reserved for Microware use only.

`dd_mode`

During `I_CREATE` or `I_OPEN` system calls, the value in this bit is used to check the validity of a caller's access mode byte. If a bit is set, the device can perform the corresponding function. The `S_ISIZE` bit is usually set, because it is handled by the file manager or ignored. If the `S_ISHARE` bit is set, the device is non-sharable. A printer is an example of a non-sharable device. The following values are defined in the header file `modes.h`:

Defined Name	Value	Description
`S_IPRM`	`0xffff`	Mask for permission bits
`S_IREAD`	`0x0001`	Owner read
`S_IWRITE`	`0x0002`	Owner write
`S_IEXEC`	`0x0004`	Owner execute
`S_ISEARCH`	`0x0004`	Search permission
`S_IGREAD`	`0x0010`	Group read
`S_IGWRITE`	`0x0020`	Group write
`S_IGEXEC`	`0x0040`	Group execute
`S_IGSEARCH`	`0x0040`	Group search
`S_IOREAD`	`0x0100`	Public read
`S_IOWRITE`	`0x0200`	Public write
`S_IOEXEC`	`0x0400`	Public execute
`S_IOSEARCH`	`0x0400`	Public search
`S_ITRUNC`	`0x0100`	Truncate on open
`S_ICONTIG`	`0x0200`	Ensure contiguous file
`S_IEXCL`	`0x0400`	Error if file exists on create
`S_ICREAT`	`0x0800`	Create file
`S_IAPPEND`	`0x1000`	Append to file
`S_ISHARE`	`0x4000`	Non-sharable

`dd_fmgr`

Offset to the name string of the file manager module to use.

`dd_drvr` Offset to the name string of the device driver module to use.

`dd_class` Used to identify the class of the device, as random or sequential access. The following values are defined in the header file `io.h`:

Defined Name	Value	Description
`DC_SEQ`	`0x0001`	Sequential access device
`DC_RND`	`0x0002`	Random access device

`dd_dscres` This field is reserved for future use.

The above offsets are offsets from the beginning address of the device descriptor module.

PATH DESCRIPTORS

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

Path descriptors are variable in size. The full RBF, SBF, SCF, and PCF path descriptor structures are provided in `rbf.h`, `sbf.h`, `scf.h`, and `pcf.h` respectively. Generally, they consist of three main sections:

- A structure common to all path descriptors: `pd_com`
- A section of elements used by IOMAN, file managers, and device drivers
- The path descriptor option section

IOMAN requires the first part to be common to all path descriptors. It uses this common section to manage accesses to the path and to dispatch to the associated file manager. File managers and device drivers can add the information they need after the common part. The options section is used to contain the dynamically alterable operating parameters for the file or device. The appropriate file manager copies the path descriptor options from the device descriptor module when a path is opened or created.

You can use the SS_PATHOPT and getstat and setstat I/O system calls to update the option section of each path descriptor. You can not update any other fields of the path descriptor. The format of the common part is defined in the header file io.h and shown here. Any data defined by specific file managers is provided in the ***OS-9 Device Descriptor and Configuration Module Reference***.

In user-state, the default setting for the maximum number of paths each process can have open at any time is 32. You can change this setting by using the _os_ioconfig system call. In system-state, the maximum number of open paths depends on available system resources. See I_CONFIG for more information.

pd_com

Declaration

```
typedef struct pathcom {
     path_id          pd_id;      /* path number */
     Dev_list         pd_dev;     /* device list element pointer */
     owner_id         pd_own;     /* path creator */
     struct pathcom   *pd_paths,  /* list of open paths on device */
                      *pd_dpd;    /* ptr to default directory path desc*/
     u_int16          pd_mode,    /* mode (READ_, WRITE_, or EXEC_) */
                      pd_count,   /* actual number of open images */
                      pd_type,    /* device type */
                      pd_class;   /* device class */
     process_id       pd_cproc;   /* current active process ID */
     u_char           *pd_plbuf,  /* pointer to partial pathlist */
                      *pd_plist;  /* pointer to complete pathlist */
     u_int32          pd_plbsz;   /* size of pathlist buffer */
     lk_desc          pd_lock;    /* reserved for internal use */
     void             *pd_async;  /* asynchronous I/O resource pointer */
     u_int32          pd_state;   /* process status bits */
     u_int32          pd_rsrv[7]; /* reserved */
} pd_com, *Pd_com;
```

Fields

pd_id

Contains the system path number of the path descriptor.

pd_dev

Pointer to the device list element of the device on which this path is opened.

pd_own

Group/user number of the process that created the path descriptor.

pd_paths

Pointer to the next path descriptor in the list of paths opened on the device.

pd_dpd

Pointer to the default directory path descriptor. When IOMAN creates a path descriptor, and a device name was not specified in the pathlist, it stores a pointer to the path descriptor for the default data or execution (as specified by the mode) directory in this field.

pd_mode

Requested access mode specified when the path descriptor is created.

pd_count

Number of users using the path. When the path descriptor is created this field is set to 1. pd_count is incremented when the path is duplicated using the I_DUP system call. The I_CLOSE request decrements this field.

pd_type

Indicates the device type. The following values are defined in the header file io.h:

Defined Name	Value	Description
DT_SCF	0	Sequential Character File Type
DT_RBF	1	Random Block File Type
DT_PIPE	2	Pipe File Type
DT_SBF	3	Sequential Block File Type
DT_NFM	4	Network File Type
DT_CDFM	5	Compact Disc File Type
DT_UCM	6	User Communication Manager
DT_SOCK	7	Socket Communication Manager
DT_PTTY	8	Pseudo-Keyboard Manager
DT_GFM	9	Graphics File Manager
DT_PCF	10	PC-DOS File Manager
DT_NRF	11	Non-volatile RAM File Manager
DT_ISDN	12	ISDN File Manager
DT_MPFM	13	MPFM File Manager
DT_RTNFM	14	Real-Time Network File Manager

Defined Name	Value	Description
`DT_SPF`	`15`	Stacked Protocol File Manager
`DT_INET`	`16`	Inet File Manager
`DT_MFM`	`17`	Real-Time Network File Manager
`DT_DVM`	`18`	Generic Device File Manager
`DT_NULL`	`19`	Null File Manager
`DT_DVDFM`	`20`	DVD File Manager
`DT_MODFM`	`21`	Module Directory File Manager

DT-Codes up to 127 reserved for Microware use only.

`pd_class` Indicates the device class. It is used to load modules. The following values are defined in the header file `io.h`:

Defined Name	Value	Description
`DC_SEQ`	`0x0001`	Serial Devices (bit 0 set)
`DC_RND`	`0x0002`	Random Access Devices (bit 1 set)

Software checking this field should test these bits only, as the rest may be defined in the future.

`pd_cproc` Process ID of the process currently using the path.

`pd_plbuf` Pointer to the partial pathlist buffer. This points to the portion of the pathlist relevant to the file manager.

`pd_plist` Pointer to the complete pathlist.

`pd_plbsz` Size of the pathlist buffer.

`pd_lock` Reserved for internal use.

`pd_async` Pointer to resources used for performing asynchronous I/O operations.

`pd_state` Process status bits used by file managers and drivers to determine the state of a process.

Defined Name	Value	Description
`PD_SYSTATE`	`0x00000001`	I/O request made from system state

`pd_rsrv` Reserved.

ACCESS MODES AND PERMISSIONS

The following sections define each of the possible access modes and permissions for applicable OS-9 I/O calls.

Access Modes

When dealing with a path, certain characteristics must be defined. The header file `<modes.h>` contains `#define` values, which correspond to bit settings of various access permissions. With a path, you typically specify whether you are going to `READ` it, `WRITE` to it, or both. Other special purpose mode settings are also available:

`FAM_READ`	Read mode. Path is open for reading.
`FAM_WRITE`	Write mode. Path is open for writing.
`FAM_EXEC`	Execute mode. Search execution directory.
`FAM_APPEND`	Append mode. Append to existing file.
`FAM_SIZE`	Initial file size specified mode.
`FAM_NONSHARE`	Non-sharable mode.
`FAM_DIR`	Directory mode.
`FAM_NOCREATE`	Do not recreate existing file mode.

Permissions

Files and named pipes also contain access permissions, which specify how the rest of the world can access the resource:

`FAP_READ`	File created with owner read permission.
`FAP_WRITE`	File created with owner write permission.
`FAP_EXEC`	File created with owner execute permission.
`FAP_GREAD`	File created with group read permission.
`FAP_GWRITE`	File created with group write permission.
`FAP_GEXEC`	File created with group execute permission.
`FAP_PREAD`	File created with world read permission.
`FAP_PWRITE`	File created with world write permission.
`FAP_PEXEC`	File created with world execute permission.

FILE MANAGERS

File managers perform the following functions:

- Process the raw data stream to or from device drivers for a class of similar devices.
- Service all of the I/O system service requests for a class of devices; those not handled by the file manager are passed to the device driver by the file manager.
- Responsible for mass storage allocation and directory processing, if applicable to the class of devices they service.
- Buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory.
- Monitor and process the data stream.

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

The following file managers are included in typical systems:

File Manager	Description
RBF (Random Block File Manager)	Operates random-access, block-structured devices such as disk systems.
SCF (Sequential Character File Manager)	Used with single-character-oriented devices such as CRT or hardcopy terminals, printers, and modems.
PIPEMAN (Pipe File Manager)	Supports interprocess communication through memory buffers called pipes.
SBF (Sequential Block File Manager)	Used with sequential block-structured devices such as tape systems.
PCF (PC File Manager)	Transfers files between OS-9 and DOS systems.
SPF (Stacked Protocol File Manager)	Manages communications. Refer to the SoftStax manual set for more information about SPF.

File Manager Organization

A file manager is a collection of major subroutines accessed through a dispatch table in the static storage of the file manager. IOMAN locates this table by adding an offset specified by the `m_share` field of the file manager module header. The table contains the starting address of each file manager subroutine. The first entry of the table contains the number of subroutines pointed to by the table.

Dispatch Table Sample Listing

Declaration

```
#include <types.h>
#define FUNC_COUNT 16

struct {
     u_int32        func_count;                 /* number of functions */
     error_code     (*funcs[FUNC_COUNT])();     /* function table */
} dispatch_table = { FUNC_COUNT,
 { Attach, Chgdir, Close, Create, Delete, Detach, Dupe, Getstat, Makdir, Open,
Read, Readln, Seek, Setstat, Write, Writeln }
};
```

Description

When IOMAN calls a file manager subroutine, it always passes two parameters. For the `Attach` and `Detach` functions, the first parameter is a pointer to the parameter block of the caller and the second is a pointer to the device list entry. For all other functions, the first parameter is the pointer to the caller's parameter block and the second is a pointer to the path descriptor for the specified path.

Functions

`Attach`

When an `I_ATTACH` call is made to a device, a file manager determines whether the device has been previously attached. If it has, the file manager increments the use count for the device and returns. If the device has not been previously attached, the file manager may perform some additional logical unit initialization and calls the init routine of the device driver to initialize the hardware.

If the device driver's init routine returns an error, the file manager returns the error.

Chgdir

On multi-file devices, I_CHDIR searches for a directory file. IOMAN allocates a path descriptor. This allows I_CHGDIR to save information about the directory file for later searches. IOMAN saves the path identifier in the I/O process descriptor.

I_OPEN and I_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 an appropriate error code.

Close

I_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.

I_CLOSE may perform specific end-of-file processing if necessary, such as writing end-of-file records on tapes.

Create

I_CREATE performs the same function as I_OPEN. If the file manager controls multi-file devices (RBF and PIPEMAN), a new file is created.

Delete

Multi-file device managers usually do a directory search similar to I_OPEN. Once the specified file is found, these managers remove the file name from the directory. Any media in use by the file is returned to unused status.

Detach

When an I_DETACH call is made to a device, a file manager decrements the use count for the device. If the count is still non-zero, the file manager returns. If the use count becomes zero, the file manager calls the driver's terminate routine. If the terminate routine returns an error, the file manager returns the error.

Dupe

IOMAN implements all of the functions of the I_DUP system call on a device. Normally, file managers are called but do nothing.

Getstat

The I_GETSTAT (get status) system calls are wildcard calls that retrieve the status of various features of a device (or file manager) that are not generally device independent.

The file manager can perform a specific function such as obtaining the size of a file. Status calls that are unknown by the file manager are passed to the driver to provide a further means of device independence.

Makdir

I_MAKDIR creates a directory file on multi-file devices. I_MAKDIR is neither preceded by a Create nor followed by a Close. File managers that cannot support directories or do not support multi-file devices should return the EOS_UNKSVC (unknown service request) error.

Open

I_OPEN opens a file on a particular device. This typically involves allocating any required buffers, initializing path descriptor variables, and parsing the path name. If the file manager controls multi-file devices (RBF and PIPEMAN), directory searching is performed to find the specified file.

Read

I_READ returns the requested number of bytes to the user's data buffer. If no data is available, an EOF error is returned. I_READ must be capable of copying pure binary data, and generally does not perform editing on the data.

Readln

I_READLN differs from I_READ in two respects. First, I_READLN is expected to terminate when the first end-of-line character (carriage return) is encountered. Second, I_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.

Seek

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

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

Setstat

The I_SETSTAT (set status) system call is the same as the I_GETSTAT function except it is generally used to set the status of various features of a device or file manager.

The I_SETSTAT and I_GETSTAT system calls are wildcard calls designed to access features of a device (or file manager) that are not generally device independent. Status calls that are unknown to the file manager are passed to the device driver.

`Write`

`I_WRITE`, like `I_READ`, must be capable of recording pure binary data without alteration. Usually, the routines for read and write are nearly identical. The most notable difference is `I_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 using fixed-length records (sectors) must often preread a sector before writing it unless the entire sector is being written.

`Writeln`

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

DEVICE DRIVER MODULES

Device driver modules perform basic low-level physical I/O functions. For example a basic function of the disk driver is to read or write a physical sector. The driver is not concerned about files and directories, which are handled at a higher level by the OS-9 file manager. Because device drivers are re-entrant, one copy of the module can simultaneously support multiple devices using identical I/O controller hardware.

This section describes the general characteristics of OS-9 device drivers. If you are developing or modifying a device driver, read the *OS-9 Porting Guide*.

Basic Functional Driver Requirements

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 a small device descriptor module.

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

A device driver is actually a package of subroutines called by a file manager in system state. Device driver functions include:

- Initializing device controller hardware and related driver variables as required
- Reading standard physical units (a character or sector depending on the device type)
- Writing standard physical units (a character or sector depending on the device type)
- Returning specified device status
- Setting specified device status
- De-initializing devices, assuming the device will not be used again unless re-initialized
- Processing error exceptions generated during driver execution

All drivers must conform to the standard OS-9 memory module format. The module type code is `MT_DEVDRVR`. 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 will require all device drivers to be system-state modules.

Interrupts and DMA

Because OS-9 is a multi-tasking operating system, optimum system performance is obtained when all I/O devices are configured for interrupt-driven operation.

- For character-oriented devices, set up the controller to generate an interrupt on receipt of an incoming character and at the completion of transmission of an out-going character. Both the input data and the output data should be buffered in the driver.
- For block-type devices (RBF and SBF), set up the controller to generate an interrupt upon the completion of a block read or write operation. The driver does not need to buffer data because the driver is passed the address of a complete buffer. A Direct Memory Access (DMA) device, if available, significantly improves the data transfer speed.

Usually, the initialization subroutine of the device driver adds the relevant device interrupt service routine to the OS-9 interrupt polling system using the `F_IRQ` system call. The controller interrupts are enabled and disabled by the data transfer routines (for example, `I_READ` and `I_WRITE`) as required. The termination subroutine disables the interrupt hardware and removes the device from the interrupt polling system.

The assignment of device interrupt 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 is an example of how you can assign interrupt levels:

```
High:    clock ticker
         "dumb" (non-buffering) disk controller
         terminal port
         printer port
Low:     "smart" (sector-buffering) disk controller
```

4 Interprocess Communication

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

- Signals
- Alarms
- Events
- Semaphores
- Pipes
- Operations on Pipes
- Data Modules

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. 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 similarly to how a CPU receives an interrupt. Signals enable a process to send a numbered interrupt to another process. If an active process receives a signal, the intercept routine is executed immediately (if installed) and the process resumes execution where it left off. If a sleeping or waiting process receives a signal, the process is moved to the active queue, the signal routine is executed, and the process resumes execution right after the call that removed it from the active queue.

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

Each signal has two parts:

- process ID of the destination
- signal code

Signal Codes

OS-9 supports the following signal codes.

Signal	Description
1	**Wake-up signal**. Sleeping/waiting processes receiving this signal are awakened, but the signal is not intercepted by the intercept handler. Active processes ignore this signal. A program can receive a wake-up signal safely without an intercept handler. The wake-up signal is not queued.
2	**Keyboard abort signal**. When <control>E is typed, this signal is sent 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.

Signal	Description
3	**Keyboard interrupt signal**. When <control>C is typed, this signal is sent 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	**Unconditional system 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.
5	**Hang-up signal**. SCF sends this signal when the modem connection is lost.
6-19	Reserved
20-25	Reserved
26-31	User-definable signals that are deadly to I/O operations.
32-127	Reserved
128-191	Reserved
192-255	Reserved
256- 4294967295	User-definable non-deadly to I/O signals.

Refer to the `_pthread_setsignalrange()` function to specify the range of signals that the Pthread layer uses. By default, the Pthreads layers use signal values between 40,000 and 49,999 inclusive.

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

The following system calls enable processes to communicate through signal.

Name	Description
F_ICPT	Installs a signal intercept routine.
F_SEND	Sends a signal to a process.
F_SIGLNGJ	Sets signal mask value and returns on specified stack image.
F_SIGMASK	Enables/disables signals from reaching the calling process.
F_SIGRESET	Resets process intercept routine recursion depth.
F_SLEEP	Deactivates the calling process until the specified number of ticks has passed or a signal is received.

Refer to the following for more information:

- For specific information about these system calls, refer to *Chapter 8, OS-9 System Calls*. The Microware Ultra C/C++ compiler also supports a corresponding C call for each of these calls.
- Refer to *Appendix A, Example Code* for a sample program demonstrating how you can use signals.

Signal Implementation

For some advanced applications, it is helpful to understand how the operating system invokes a signal intercept routine when delivering a signal to a process. It may be necessary to understand the contents of the user stack when executing a process’ signal intercept routine. An application can call a signal intercept routine either non-recursively or recursively.

Non-recursive Calling

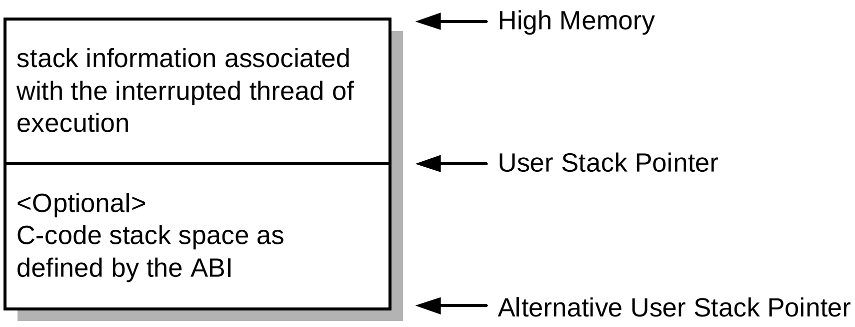
When trying to synchronize signals, most applications call signal intercept routines for a process non-recursively. In the case of non-recursive invocation of the intercept routine, the operating system performs the following tasks to maintain the user stack for the process:

1. Save the process' main executing context on the process' system state stack.
2. Loads the process' global statics pointer associated with the intercept routine (as specified when performing the `F_ICPT` call).
3. Loads the process' code constant pointer.
4. Loads the process' user stack pointer with its value at the time of the signal interruption.
5. Calls the process' intercept routine.

In some cases, depending on the target system, the C-code application binary interface (ABI) can require the operating system allocate some additional stack space in order to call a C-code intercept routine.

Figure 4-1 shows the user stack contents as it appears in the case of a non-recursive invocation of a signal intercept routine.

Figure 4-1. Non-recursive Invocation of Signal Intercept Routine


stack information associated with the interrupted thread of execution
<Optional>
C-code stack space as defined by the ABI
High Memory
User Stack Pointer
Alternative User Stack Pointer

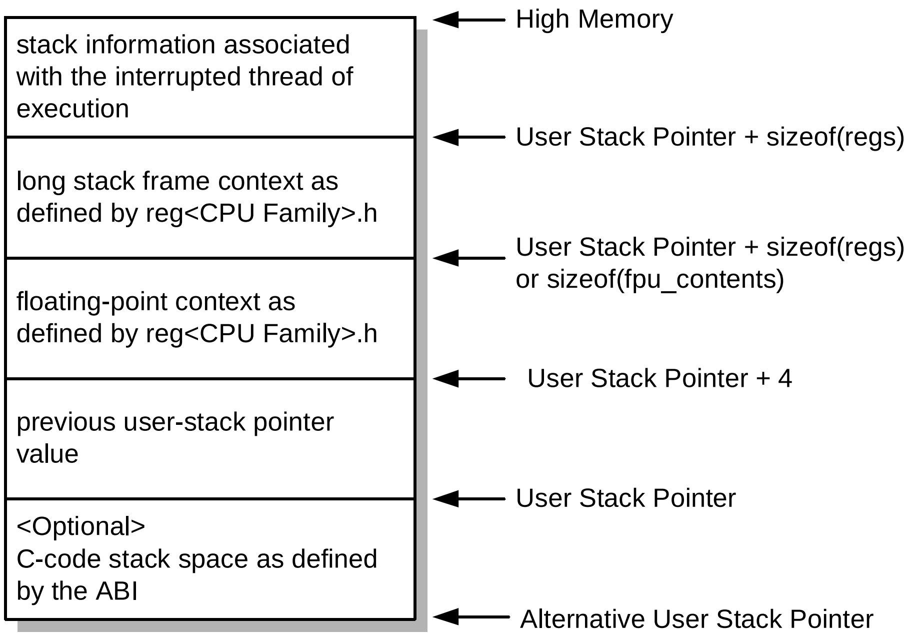
Recursive Calling

Normally, the operating system prevents recursive invocation of an intercept routine by incrementing a variable associated with the process, known as the signal mask, when calling the intercept routine. The operating system then decrements the signal mask value upon returning from the intercept routine through the `F_RTE` system call. When the operating system sees that the signal mask of a process is non-zero, it does not attempt to invoke the intercept routine when it detects a pending signal.

The only way an intercept routine can be called recursively when a signal is pending is if the process explicitly clears its signal mask, through the `F_SIGMASK` or `F_SIGLNGJ` system calls, or implicitly via the user-state `F_SLEEP` and `F_WAIT` services, from within the context of its intercept routine. When calling an intercept routine recursively, the stack contents of the user stack are quite different from the non-recursive case. In order to keep from over consuming the system stack when saving its context, the operating system copies the saved context along with its floating-point context to the user-state stack.

Figure 4-2 shows the user-state stack contents as it appears in the case of a recursive invocation of a signal intercept routine.

Figure 4-2. Recursive Invocation of Signal Intercept Routine


stack information associated with the interrupted thread of execution
long stack frame context as defined by reg<CPU Family>.h
floating-point context as defined by reg<CPU Family>.h
previous user-stack pointer value
<Optional>
C-code stack space as defined by the ABI
High Memory
User Stack Pointer + sizeof(regs)
User Stack Pointer + sizeof(regs)
or sizeof(fpu_contents)
User Stack Pointer + 4
User Stack Pointer
Alternative User Stack Pointer

The exact contents of the floating-point context shown in Figure 4-2 can vary within a given processor family, depending on whether or not the processor has hardware support for floating point calculations. If the processor has a hardware floating-point unit (FPU), the contents of the FPU context directly reflect the hardware context. If the processor does not have a hardware FPU, the FPU context area shown in Figure 4-2 contains whatever the FPU software emulation module must preserve on behalf of the process. The actual size of this area can be determined at execution time by consulting the variable `d_fpusize` in the operating system globals area (see `F_GETSYS`).

The PowerPC 6xx series processors containing a full hardware floating-point implementation are the only processors that vary from this described stack format. For this family of processors the FPU context is actually a part of the long stack frame as described in the `regppc.h` header file. The stack format resembles the format described previously with the exception that the FPU context is not separate from the long stack format.

ALARMS

User-state Alarms

The user-state alarm requests enable a program to arrange for a signal to be sent to itself. The signal may be sent at a specific time of day or after a specified interval has passed. The program may also request the signal be sent periodically, each time the specified interval has passed.

Alarm	Description
`F_ALARM, A_ATIME`	Sends a signal at a specific time.
`F_ALARM, A_CYCLE`	Sends a signal at the specified time intervals.
`F_ALARM, A_DELET`	Removes a pending alarm request.
`F_ALARM, A_RESET`	Resets an existing alarm request.
`F_ALARM, A_SET`	Sends a signal after the specified time interval.

Cyclic Alarms

A cyclic alarm provides a time base within a program. This simplifies the synchronization of 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 are using an OS-9 process to update the real-time display of a car's digital dashboard.

The process might perform the following functions:

- 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.

Each function the process must monitor can have 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 operating system handles 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—executing programs at specific days and times. Another use is to generate a traditional alarm clock buzzer for personal reminders.

This type of alarm is sensitive to changes made to the system time. For example, assume the current time is 4:00 and a program sends itself a signal at 5:00. The program can 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 this type of 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. This can be accomplished by sending a keyboard abort signal at the maximum allowable time and then issuing 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. The example program deton.c (in *Appendix A, Example Code*) demonstrates this technique.

System-State Alarms

A system-state counterpart exists for user-state alarm function. 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:

Alarm	Description
F_ALARM, A_ATIME	Executes a subroutine at a specified time
F_ALARM, A_CYCLE	Executes a subroutine at specified time intervals
F_ALARM, A_DELET	Removes a pending alarm request
F_ALARM, A_RESET	Resets an existing alarm request
F_ALARM, A_SET	Executes a subroutine after a specified time interval

The alarm is executed by the kernel process, not by the original requester process. During execution, the user number of the system process is temporarily changed to the original requester. The stack pointer passed to the alarm subroutine is within the system process descriptor and contains about 4KB of free space.

The kernel automatically deletes the pending alarm requests belonging to a process when that 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 on behalf of the I/O process. This alarm does not work if it is removed when the process exits.

The alarm has persistence if the `TH_SPOWN` bit in the alarm call's `flags` parameter is set. This causes the alarm to be owned by the system process rather than the current process.

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

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`, `F_ACQLK`, `F_WAITLK`, 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 available CPU time.

For a program demonstrating how alarms can be used, see *Appendix A, Example Code*.

EVENTS

OS-9 events are multiple value semaphores. They synchronize concurrent processes that are 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 process at a time updates the data module.

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

An OS-9 event is a global data structure maintained by the system. The event structure is listed here and is defined in the header file `events.h`. The following section contains descriptions of each field.

ev_str/ev_infostr

Event Structure

Declaration

```
typedef struct {
    event_id     ev_id;          /* event id number */
    u_int16      ev_namsz;       /* size of memory to allocate for name */
    u_char       *ev_name;       /* pointer to event name */
    u_int16      ev_link,        /* event use count */
                 ev_perm;        /* event permissions */
    owner_id     ev_owner;       /* event owner (creator) */
    int16        ev_winc,        /* wait increment value */
                 ev_sinc;        /* signal increment value */
    int32        ev_value;       /* current event value */
    Pr_desc      ev_quen,        /* next event in queue */
                 ev_quep;        /* previous event in queue */
    u_char       ev_resv[14];    /* reserved */
} ev_str, *Ev_str;
```

The structure used by the `F_EVENT, EV_INFO` request contains a subset of the standard event fields. This structure is listed here and defined in the header file `events.h`.

```
typedef struct {
    event_id     ev_id;          /* event id number */
    u_int16      ev_link,        /* event use count */
                 ev_perm;        /* event permissions */
   owner_id     ev_owner;      /* event owner (creator) */
    int16       ev_winc,        /* wait increment value */
                ev_sinc;        /* signal increment value
*/
    int32        ev_value;       /* current event value */
} ev_infostr, *Ev_infostr;
```

Description

The OS-9 event system provides the following 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

Fields

`ev_id` A unique ID is created from this number and the event's array position.

`ev_namsz` Size of the event name in bytes.

`ev_name` The event name must be unique.

`ev_link` The event use count.

`ev_perm` The event's access permissions which are used to verify that a process has access to an event when an `F_EVENT, EV_LINK` operation is performed.

`ev_owner` The ID of the event owner (creator).

`ev_winc` The event wait increment. `ev_winc` 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.

`ev_sinc` The event's signal increment. `ev_sinc` is added to the event value when the event is signaled. It is set when the event is created and does not change.

`ev_value` This four byte integer represents the current event value.

`ev_quen` 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.

`ev_quep` A pointer to the previous process in the event queue.

`ev_resv` Reserved for future use.

Wait and Signal Operations

The two most common operations performed on events are wait and signal.

Wait

The wait operation performs the following three functions:

1. Suspends the process until the event is within a specified range
2. Adds the wait increment to the current event value
3. Returns control to the process just after the wait operation was called

Signal

The signal operation performs the following three functions:

1. Adds the signal increment to the current event value
2. Checks for other processes to awaken
3. Returns control to the process

These operations enable a process to suspend itself while waiting for an event and to reactivate when another process signals the event has occurred.

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

- Wait for the resource to become available.
- Mark the resource as busy.
- Use the resource.
- Signal the resource is no longer busy.

Due to time slicing, the first two steps in this process must be indivisible. Otherwise, two processes might check an event and find it free. Then, both processes try to mark it busy. This would correspond 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 example, you can use events to synchronize the use of a printer. You set the initial event value to 0, the wait increment to -1, and the signal increment to 1. When a process wants exclusive use of the printer, it performs an event wait call with a value range of zero and checks to see if a printer is available. If the event value is zero, it applies the wait increment (-1), causing the event value to go to -1 and marking the printer as busy; the process is allowed to use the printer. A negative event value indicates the printer is busy; the process is suspended until the event value comes into range (becomes zero in this case). When a process is finished with the printer, it performs an event signal call, the signal increment is applied causing the event value to be incremented by one, and then the process in range is activated.

For a program demonstrating how events can be used see *Appendix A, Example Code*.

The F_EVENT System Call

The `F_EVENT` system call creates named events for this type of application. The name event was chosen instead of semaphore because `F_EVENT` synchronizes processes in a variety of ways not usually found in semaphore primitives. OS-9 event routines are very efficient and are 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.

Functions exist to enable you to create, delete, link, unlink, and examine events. Several variations of the signal and wait operations are also provided. Specific parameters and functions of each event operation are discussed in the `F_EVENT` description in *Chapter 8, OS-9 System Calls*. The following event functions that are supported:

Event	Description
`F_EVENT, EV_ALLCLR`	Wait for all bits defined by mask to become clear.
`F_EVENT, EV_ALLSET`	Wait for all bits defined by mask to become set.
`F_EVENT, EV_ANYCLR`	Wait for any bits defined by mask to become clear.

Event	Description
F_EVENT, EV_ANYSET	Wait for any bits defined by mask to become set.
F_EVENT, EV_CHANGE	Wait for any of the bits defined by mask to change.
F_EVENT, EV_CREAT	Create new event.
F_EVENT, EV_DELET	Delete existing event.
F_EVENT, EV_INFO	Return event information.
F_EVENT, EV_LINK	Link to existing event by name.
F_EVENT, EV_PULSE	Signal an event occurrence.
F_EVENT, EV_READ	Read event value without waiting.
F_EVENT, EV_SET	Set event variable and signal an event occurrence.
F_EVENT, EV_SETAND	Set event value by ANDing the event value with a mask.
F_EVENT, EV_SETOR	Set event value by ORing the event value with a mask.
F_EVENT, EV_SETR	Set relative event variable and signal an event occurrence.
F_EVENT, EV_SETXOR	Set event value by XORing the event value with a mask.
F_EVENT, EV_SIGNL	Signal an event occurrence.
F_EVENT, EV_TSTSET	Wait for all bits defined by mask to clear; set these bits.
F_EVENT, EV_UNLNK	Unlink event.
F_EVENT, EV_WAIT	Wait for event to occur.
F_EVENT, EV_WAITR	Wait for relative to occur.

SEMAPHORES

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

Since using C bindings is the preferred method of accessing OS-9 semaphores, `F_SEMA` is not documented in Chapter 8. See the *Ultra C/C++ Library Reference* for information on the `os_sema` calls.

OS-9 supports the semaphore routines shown in the following table:

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.

A single semaphore system call, `F_SEMA`, provides all of the semaphore functionality. `F_SEMA` requires the following 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. In addition, you must be certain that this storage is properly initialized (to a known value) prior to initializing the semaphore structure for use (via `_os_sema_init()`). In many cases this storage initialization is handled for you by OS-9 (such as inside a data module or part of global data).

For a program demonstrating how you may use semaphores, see *Appendix A, Example Code*.

A typical application using semaphores might 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 OS-9 module protection mechanisms to protect the semaphore.

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 — When a semaphore is reserved, any process attempting to reserve the semaphore waits. This includes the process that has the semaphore reserved.

Free — When a semaphore is 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 wait queue of the semaphore.

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 queue is activated and retries the reserve operation on the semaphore.

The definition for the semaphore structure can be found in the `semaphore.h` header file. Semaphores use the following data structure:

```
/* Semaphore structure definition */
typedef struct semaphore {
    sema_val
              s_value;           /* semaphore value (free/busy status) */
    u_int32   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 (process ID) */
              s_user,            /* reserved for users       */
              s_flags,           /* general purpose bit-field flags */
              s_sync,            /* integrity sync code */
              s_reserved[3];     /* reserved for system use */
} semaphore, *Semaphore;
```

PIPES

An OS-9 pipe is a first-in first-out (FIFO) buffer that enables 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`.

A pipe contains 128 bytes, unless a different buffer size is specified when the pipe is created. Typically, a pipe is used as a one-way data path between two processes:

- Writing
- 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 send data to itself. You can use this to simplify type conversions by printing data into the pipe and reading it back using a different format.

Data transfer through pipes is extremely efficient and flexible. Data does not have to be read out of the pipe in the same size sections in which it was written.

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

- Longer messages (more than 32 bits)
- Queued messages
- Determination of pending messages
- Easy process-independent coordination (using named pipes)

Named and Unnamed Pipes

OS-9 supports both named and unnamed (anonymous) pipes. The shell uses unnamed pipes extensively to construct program *pipelines*, but user programs can also use them. Unnamed pipes can be opened only once. Independent processes may communicate through them only if the pipeline was constructed by a common parent to the processes. This is accomplished by making each process inherit the pipe path as one of its standard I/O paths.

The use of named pipes is similar to that of unnamed pipes. The main difference is a named pipe can be opened by several independent processes, which simplifies pipeline construction. Other specific differences are noted in the following sections.

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.

You can 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 (`EOS_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 can specify access permissions. They are handled similarly to permissions on files in random block file systems.

The size of the default FIFO buffer associated with a pipe is specified in the pipe device descriptor. To override this default when creating a pipe, set the initial file size bit of the mode parameter and pass the desired file size in the parameter block.

If no default or overriding size is specified, a 128-byte FIFO buffer is created.

You can rename a named pipe to an unnamed pipe and an unnamed pipe to a named pipe.

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.

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 through the inheritance of the standard I/O paths through the `F_FORK` call. As an example, the following C language pseudocode outline describes a method the shell can use to construct a pipeline for the command `dir -u ! qsort`. It is assumed paths 0 and 1 are already open.

```
StdInp =     _os_dup(0)        save the shell’s standard 
input
StdOut =    _os_dup(1)       save shell’s standard output
             _os_close(1)      close standard output
          _os_open("/pipe")     open the pipe (as path 1)
          _os_fork("dir","-u")  fork "dir" with pipe as 
standard output
          _os_close(0)          free path 0
          _os_dup(1)            copy the pipe to path 0
          _os_close(1)          make path available
          _os_dup(StdOut)       restore original standard 
out
          _os_fork("qsort")     fork qsort with pipe as 
standard input
          _os_close(0)          get rid of the pipe
          _os_dup(StdInp)       restore standard input
          _os_close (StdInp)    close temporary path
          _os_close (StdOut)    close temporary path
```

The main advantage of using named pipes is several processes can communicate through the same named pipe without having to inherit it from a common parent process. For example, the above steps can be approximated by the following command:

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

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

Read/Readln

The `I_READ` and `I_READLN` system calls return the next bytes in the pipe buffer. If not enough data is ready to satisfy the request, the process reading the pipe is put to sleep until more data becomes available.

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 the end-of-file was reached, an end-of-file error is not returned. However, the returned byte count is the number of bytes actually transferred, which is less than the number requested.

The read and write system calls are faster than the `readln` and `writeln` system calls because PIPEMAN does not have to check for carriage returns and the loops moving data are tighter.

Write/Writeln

The I_WRITE and I_WRITELN system calls work in almost the same way as I_READ and I_READLN. A pipe error (EOS_WRITE) is returned when all the processes with a full unnamed pipe open attempt to write to the pipe. Since there is no reader process, 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 EOS_WRITE error. If a named pipe becomes full before a process receiving data from the pipe has opened it, the process writing to the pipe is put to sleep until a process reads the pipe.

Close

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

Getstat/Setstat

PIPEMAN supports a wide range of status codes enabling the insertion of pipes as a communications channel between processes where an random block file (RBF) or serial character file (SCF) device would normally be used. For this reason, most RBF and SCF status codes are implemented to perform without returning an error. The actual function may differ slightly from the other file managers, but it is usually compatible.

GetStat Status Codes Supported by PIPEMAN

The following table shows only the supported GetStat status codes. All other codes return an EOS_UNKSVC error (unknown service request).

Name	Description
I_GETSTAT, SS_DEVOPT	Read the default path options for the device.
I_GETSTAT, SS_EOF	Test for end-of-file condition.
I_GETSTAT, SS_FD	Read the pseudo file descriptor image for the pipe associated with the specified path.

Name	Description
I_GETSTAT, SS_FDINFO	Read the pseudo file descriptor sector for the pipe specified by a sector number.
I_GETSTAT, SS_LUOPT	Read the logical unit options section.
I_GETSTAT, SS_PATHOPT	Read the path options section of the path descriptor.
I_GETSTAT, SS_READY	Test whether data is available in the pipe. It returns the number of bytes in the buffer.
I_GETSTAT, SS_SIZE	Return the size of the associated pipe buffer.

SetStat Status Codes Supported by PIPEMAN

The following table shows the SetStat status codes supported by PIPEMAN.

Name	Description
I_SETSTAT, SS_ATTR	Changes the file attributes of the associated pipe.
I_SETSTAT, SS_DEVOPT	Does nothing, but returns without error.
I_SETSTAT, SS_FD	Writes the pseudo file descriptor image for the pipe.
I_SETSTAT, SS_LUOPT	Does nothing, but returns without error.
I_SETSTAT, SS_PATHOPT	Does nothing, but returns without error.
I_SETSTAT, SS_RELEASE	Releases the device from the SS_SENDSIG processing before data becomes available.
I_SETSTAT, SS_RENAME	Changes the name of a named pipe, changes a named pipe to an unnamed pipe, and changes an unnamed pipe to a named pipe.
I_SETSTAT, SS_SIZE	Resets the pipe buffer if the specified size is zero. Otherwise, it has no effect, but returns without error.
I_SETSTAT, SS_SENDSIG	Sends the process the specified signal when data becomes available.

The I_MAKDIR and I_CHDIR service requests are illegal service routines on pipes. They return EOS_UNKSVC.

Pipe Directories

Opening an unnamed pipe in the Dir mode enables 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 enables utilities that normally read an RBF directory file sequentially to work with pipes.

PIPEMAN is not a true directory device, so commands like chd and makdir do not work with /pipe.

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

For example, if two pipe descriptors exist, a directory of either device reveals all the named pipes for both devices. If each pipe descriptor has a unique port address (0, 1, 2, etc.) or unique logical unit number, the I/O system allocates different logical unit static storage for each pipe device. This produces expected results.

DATA MODULES

OS-9 data modules enable multiple processes to share a data area and to transfer data among themselves. A data module must have a module header and a valid CRC to be loaded into memory. Data modules can be non-reentrant (modifiable). One or more processes can share and modify the contents of a data module.

OS-9 does not have restrictions as to the content, organization, or use of the data area in a data module. These considerations are determined by the processes using the data module.

OS-9 does not synchronize processes using a data module. Consequently, thoughtful programming, usually involving events or signals, is required to enable 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 and entered into the calling process' current module directory. A CRC value is not computed for the data module when it is created.

It is essential the data module 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()` and `_os_datmod()` calls which are specific to data modules, and the `modlink()`, `modload()`, `munlink()`, `munload()`, `_os_link()`, `_os_unlink()`, `_os_unload()`, `_os_setcrc()`, and `_setcrc()` calls that apply to all OS-9 modules.

For more information on these calls, refer to the *Using Ultra C/C++* manual.

The 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 0. If you want the module to remain in memory when the link count is zero, make the module sticky by setting the sticky bit in the module header attribute byte.

Saving to Disk

If a data module is saved to disk, you can use the `dump` utility to examine the module format and contents. You can save a data module to disk with the `save` utility or by writing the module image into a file. If the data module was modified since its CRC value was created, the saved module CRC will be bad and it becomes impossible to reload the module into memory.

To allow the module to be reloaded, use the F_SETCRC system call or the `_setcrc()` C library call before writing the module to disk. Or, use the `fixmod` utility after the module has been written to disk.

5 Subroutine Libraries and Trap Handlers

This chapter explains how to install, execute, and terminate subroutine libraries. It also explains how to install and execute trap handlers. It includes the following topics:

- Subroutine Libraries
- Trap Handlers

SUBROUTINE LIBRARIES

An OS-9 subroutine library is a module containing a set of related or frequently used subroutines. Subroutine libraries enable distinct processes to share common code. Any user program may dynamically link to the user subroutine library and call it at execution time.

Although subroutine libraries reduce the size of the execution program, they do not accomplish anything that could not be done by linking the program with the appropriate library routines at compilation time. In fact, programs calling subroutine libraries execute slightly slower than linked programs performing the same function. A program can link to a maximum of sixteen subroutine libraries, numbered from zero to fifteen.

Microware provides a standard subroutine library of I/O conversions for C language programs. Subroutine library identifier zero is reserved for the Microware `csl` subroutine library.

Like standard OS-9 program modules, subroutine libraries have one entry point and may have their own global static storage. The module type of subroutine library modules is `MT_SUBROUT` and the module language is `ML_OBJECT`.

Subroutine functions are usually executed as though they were called directly by the main program. System calls or other operations that could be performed by the calling module can also be performed in a subroutine library.

Installing and Executing Subroutine Libraries

To install a subroutine library, a user program must use the `F_SLINK` system call. `F_SLINK` attempts to link to the subroutine library. If the link is successful, it allocates and initializes the global static storage and returns pointers to the library's entry point and to the library's global static storage area.

Typically, a main program's first call to a subroutine library calls an initialization routine. The initialization routine usually has very little to do, but could be used to open files, link to additional subroutine libraries or data modules, or perform other startup activities.

The main program must save the entry pointer and static storage pointer returned by `F_SLINK` to enable subsequent calls to the subroutine library.

The OS-9 C library provides functions to install and call subroutine libraries. The `_sliblink()` function installs a specified subroutine module saving the subroutine library's entry and global static storage pointers in the global arrays `_sublibs[]` and `_submems[]`, respectively.

You can use the `_subcall` function to call an existing subroutine library. For example, suppose the main program reference in C is the following statement:

```
my_function(p1, p2, p3, p4)
```

The `_subcall` reference in 80386 assembler would be as follows:

```
my_function: call _subcall
dc.l SUB_LIB_NUM
dc.l SUB_MY_FUNCTION
```

`_subcall` does the following:

- Retrieves the subroutine library and function identifiers
- Adjusts the program stack
- Dispatches to the subroutine library entry point with the correct global static storage configuration

The return from the subroutine in the subroutine library takes the flow of execution directly back to the initial function reference in the main program.

To create a subroutine library, you must create a table of `_subcall` calls, and subroutine library and function identifiers as previously described. In addition, some dispatch code must be written in the subroutine library. For more information, refer to the subroutine library example provided in the Subroutine Library section of *Appendix A, Example Code*.

Terminating Subroutine Libraries

Programs using subroutine libraries do not need to explicitly terminate the use of the libraries. When a process terminates, the OS-9 kernel unlinks any subroutine libraries and releases their resources on behalf of the process. But, a program may terminate the use of a subroutine library explicitly by performing a `_sliblink()` call. In this case, you must specify a null string for the subroutine library name and the associated subroutine library identifier. This unlinks the subroutine library and returns its resources to the system.

These are the resources associated with the calling process' invocation of the subroutine library and do not affect the resources of other processes using the same subroutine library.

TRAP HANDLERS

Trap handlers are similar to subroutine libraries with the following exceptions:

- When a trap handler is linked, the kernel calls the trap initialization entry point. The kernel does not call an initialization entry point when the subroutine library is linked. Instead, the main program must call the initialization routine, if one exists.
- A trap handler may have more than one entry point; there is exactly one entry point in a subroutine library.
- Trap handlers only execute in system state; subroutine libraries execute in the same state as the main program.
- There may be a termination routine for a trap handler; there is no explicit termination entry point for a subroutine library.
- Dispatching to subroutine libraries does not involve the kernel in any way.

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 `MT_TRAPLIB` and module language `ML_OBJECT`.

The trap module routines are usually executed as though they were called with the standard function call instruction, except for minor stack differences. Any system calls or other operations that could be performed by the calling module are usable in the trap module.

An example C trap handler is included in *Appendix A, Example Code*.

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 performs the following functions:

- Links to the trap module
- Allocates and initializes its static storage, if any
- Executes the trap module’s initialization routine

Typically, the initialization routine has very little to do. It can open files, link to additional trap or data modules, or 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 initialization code. At initialization, a particular trap number (0 - 15) is specified that refers to the trap vector.

The OS-9 relocatable macro assembler has a special mnemonic (`tcall`) for making trap library function calls. The syntax for the `tcall` mnemonic is as follows:

```
tcall <trap library number>, <function code>
```

Usually, a table of `tcalls` with associated labels is created for calling the trap library functions from C programs. For example:

```
_asm (“
   func1: tcall T_TrapLib1, T_func1
   func2: tcall T_TrapLib1, T_func2
   .
   .
   .
   funcN: tcall T_TrapLib1, T_funcN
“);
```

Then, the main program can call the functions in the trap library as follows:

```
func1(param1, param2, ..., paramN);
```

The `tcall` mnemonic causes the program to dispatch the OS-9 kernel similarly to a system service request. The OS-9 kernel then uses the trap library identifier to dispatch to the associated trap handler module.

To create a trap handler library, you should create a table of `tcall` calls with trap handler and function identifiers as previously described. In addition, some dispatch and function return codes must be written in the trap handler module.

For more information, refer to the trap handler example provided in *Appendix A, Example Code*.

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. The program is aborted 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.

6 OS-9 File System

This chapter describes the OS-9 disk system file structure, record locking, and file security. It includes the following topics:

- Disk File Organization
- Raw Physical I/O on RBF Devices
- Record Locking
- Record Locking Details for I/O Functions
- File Security

DISK FILE ORGANIZATION

RBF supports a tree-structured file system. The physical disk organization is designed to do the following:

- Use disk space efficiently
- Resist accidental damage
- Access files quickly

This system also has the advantage of relative simplicity.

Basic Disk Organization

OS-9 supports block sizes ranging from 256 bytes to 32768 bytes in powers of two. If a disk system is used that cannot directly support the specified block size, the driver module must divide or combine blocks to simulate the allowed size.

Disks are often physically addressed by track number, surface number, and block number. To eliminate hardware dependencies, OS-9 uses a logical block number (LBN) to identify each block without regard to track and surface numbering.

It is the responsibility of the disk driver module or the disk controller to map logical block numbers to track/surface/block addresses. The OS-9 file system uses LBNs from 0 to (*n* - 1), where *n* is the total number of blocks on the drive.

All block addresses discussed in this section refer to LBNs.

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

Every OS-9 disk has the same basic structure. An identification block is located in logical block zero (LBN 0). It describes the physical and logical format of the storage volume (disk media). Each volume also includes a disk allocation map—indicating the free and allocated disk blocks, and a root directory. The identification block contains block offsets to the file descriptors of the disk allocation map and root directory.

Identification Block

LBN zero always contains the following identification block. In addition to a description of the physical and logical format of the disk, the identification block contains the volume name, date and time of creation, and additional information. If the disk is a bootable system disk, it also includes the starting LBN and size of the sysboot file.

```
typedef struct idblock {
      u_int32    rid_sync,        /* ID block sync pattern */
                 rid_diskid,      /* disk ID number (pseudo random) */
                 rid_totblocks;   /* total blocks on media */
      u_int16    rid_cylinders,   /* number of cylinders */
                 rid_cyl0size     /* cylinder 0 size in blocks */
                 rid_cylsize,     /* cylinder size in blocks */
                 rid_heads,       /* number of surfaces on disk */
                 rid_blocksize,   /* the size of a block in bytes */
                 rid_format,      /* disk format
                           Bit 0: 0 = single side
                                  1 = double side
                           Bit 1: 0 = single density
                                  1 = double density
                           Bit 2: 0 = single track (48 TPI)
                                  1 = double track (96 TPI) */
                 rid_flags,       /* various flags */
                 rid_unused1;     /* 32 bit padding */
      u_int32    rid_bitmap,      /* block offset to bitmap FD */
                 rid_firstboot,   /* block offset to debugger FD */
                 rid_bootfile,    /* block offset to bootfile FD */
                 rid_rootdir;     /* block offset to root directory FD */
      u_int16    rid_group,       /* group owner of media */
                 rid_owner;       /* owner of media */
      time_t     rid_ctime,       /* creation time of media */
                 rid_mtime;       /* time of last write to ID block */
      char       rid_name[32],    /* volume name */
                 rid_endflag,     /* big/little endian flag */
                 rid_unused2[3];  /* long word padding */
      u_int32    rid_parity;      /* ID block parity */
} idblock, *Idblock;
```

Allocation Map

The allocation map indicates which blocks have been allocated to files and which are free. Each bit in the allocation map represents a block on the disk. This means the allocation map varies in size according to the number of bits required to represent the system. If a bit is set, the block is either in use, defective, or nonexistent. rid_bitmap specifies the location of the allocation map file descriptor.

Root Directory

The root directory is the parent directory of all other files and directories on the disk. This directory is accessed using the physical device name (such as `/d1`). The location of the root directory file descriptor is specified in `rid_rootdir`.

Basic File Structure

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

All files have a file descriptor block or FD. An FD contains a list of the data segments with their starting LBNs and sizes. This is also where information such as file attributes, owner, and time of last modification is stored.

The OS-9 segmentation method keeps file data blocks 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 enables OS-9 to optimize its storage allocation.

The file descriptor structure is made up of one or more physical blocks on the disk. Only extremely large or fragmented files use more than one file descriptor block. The last element in a file descriptor is a pair of links, one to the previous file descriptor block and one to the next file descriptor block. The end of the file descriptor list is indicated by a next pointer pointing to the first or *root* file descriptor block. The information section of the file descriptor block is only valid in the root file descriptor block. Only the system uses the file descriptor structure; you cannot directly access the file descriptor.

fd_stats

The following structure, defined in the header file `rbf.h`, describes the contents of a file descriptor block.

Declaration

```
typedef struct fd_stats {
      u_int32     fd_sync,      /* file descriptor sync field */
                  fd_parity,    /* validation parity */
                  fd_flag;      /* flag word */
      u_int16     fd_host,      /* file host owner */
                  fd_group,     /* file group number */
                  fd_owner,     /* file owner number */
                  fd_links;     /* number of links to FD */
      u_int32     fd_size;      /* size of file in bytes */
      time_t      fd_ctime,     /* creation timestamp */
                  fd_atime,     /* last access timestamp */
                  fd_mtime,     /* last modified timestamp */
                  fd_utime,     /* last changed timestamp */
                  fd_btime;     /* last backup timestamp */
      u_int16     fd_rev,       /* RBF revision that created the FD */
                  fd_unused;    /* spare */
} fd_stats;
```

Fields

`fd_sync`

Identifies this block as a file descriptor block. It is set to `0xfdb0b0fd`.

`fd_parity`

Contains a 32-bit vertical parity value for the file descriptor block. It is always updated to validate the file descriptor block contents, whether in memory or on disk, to ensure the accuracy of the file structure.

`fd_flags` Contains the attributes and permissions of the file.

Flag	Description
FD_SMALLFILE	File is small enough to fit in the file descriptor
FD_DIRECTORY	File is a directory
FD_EXCLUSIVE	Only one active open allowed
PERM_OWNER_READ	Read permission by owner
PERM_OWNER_WRITE	Write permission by owner
PERM_OWNER_SRCH	Search permission by owner
PERM_OWNER_EXEC	Execute permission by owner
PERM_GROUP_READ	Read permission by group
PERM_GROUP_WRITE	Write permission by group
PERM_GROUP_SRCH	Search permission by group
PERM_GROUP_EXEC	Execute permission by group
PERM_WORLD_READ	Read permission by world
PERM_WORLD_WRITE	Write permission by world
PERM_WORLD_SRCH	Search permission by world
PERM_WORLD_EXEC	Execute permission by world

All bits not defined above are reserved

`fd_host` Contains the host owner number of the user to which the file belongs `fd_group` Contains the group number of the user to which the file belongs. This is initially set to the group number of the process creating the file. Only the owner of the file or a super user can change the group number

`fd_owner` Contains the owner number of the user to which the file belongs. This is initially set to the owner number of the process creating the file. Only the owner of the file or a super user can change the owner number

`fd_links` Contains the number of hard links to this file. A hard link is a directory entry pointing to this file

`fd_size` Contains the size of the file in bytes

`fd_ctime` Contains a time stamp representing the time when the file descriptor was initially created. This time stamp is never changed

`fd_atime` Contains a time stamp representing the time when the file was last accessed. This time stamp is updated whenever the file is opened, read, or written. If the file is a directory file, this field is not updated when it is searched by RBF

`fd_mtime` Contains a time stamp representing the time when the file was last modified. The time stamp is updated whenever a file is opened for write or a write is performed on the file

`fd_utime` Contains a time stamp representing the time when the file was last changed. The time stamp is updated whenever a write is performed on the file or the file descriptor data changes

`fd_btime` Contains a time stamp representing the last time a back up of the file was made. The backup program (`fsave`) updates the time stamp whenever a back up of the file is made

fd_rev	Contains the edition number of the RBF file manager that created the file descriptor
fd_unused	Reserved

The remainder of the file descriptor block up to the last eight bytes is filled with segment descriptors, unless the file is a **small file**. Refer to the Small Files section for details about small files.

The number of segment descriptors in the file descriptor block depends on the logical block size. The structure of a segment descriptor is shown here and defined in the header file rbf.h. The seg_offset field contains the LBN of the first block in this segment and the seg_count field contains the number of logical blocks in the segment.

```
typedef struct fd_segment {
    u_int32    seg_offset,     /* segment block offset */
               seg_count;      /* segment block count */
} fd_segment;
```

The last part of the file descriptor block contains links to other file descriptors for a file. If there is only one file descriptor for the file, these fields point to the one file descriptor block. The links structure is shown here and defined in the header file rbf.h.

```
typedef struct fd_links {
     u_int32     link_prev,     /* previous fd block */
                 link_next;     /* next fd block */
} fd_links;
```

Small Files

OS-9 RBF implements a class of files called small files. A file is considered small when its contents fit in the area of the file descriptor reserved for segments. A small file has the FD_SMALLFILE bit set in the fd_flag field. From a user's perspective, small files behave exactly like other files. RBF automatically changes a small file to a non-small file if the file grows too big to fit in the file descriptor block.

Logical Block Numbers

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

You do not have to be concerned with physical blocks. 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 block number is packaged in the file descriptor block. This makes the OS-9 record-locking functions very efficient.

Segment Allocation

Each device descriptor module has a value called a segment allocation size, that specifies the minimum number of blocks to allocate to a new segment. Set this value so file expansions do not produce a large number of tiny segments. If the system uses a small number of large files, set this field to a relatively high value, and vice versa.

When a file is created, it has no data segments allocated. Write operations past the current end-of-file allocate additional blocks to the file. The first write is always past the end-of-file. Generally, subsequent file expansions are also made in minimum allocation increments.

An attempt is made to expand the last segment before adding a new segment.

If all of the allocated blocks are not used when the file is closed, the segment is truncated and any unused blocks are deallocated in the bitmap. For random-access databases that expand frequently by only a few records, the segment list rapidly fills 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. All programs dealing with a file in write or update mode must not close the file while at end-of-file, or the file loses its excess space. The easiest way to ensure this is to perform 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 structure as other files, except the logical contents of a directory file conform to the following conventions:

- A directory file consists of an integral number of 64-byte entries.
- The end of the directory is indicated by the normal end-of-file.
- Each entry consists of a field for the file name and a field for the address of the first file descriptor block of the file.

The structure of a directory entry is shown here and defined in the header file `rbf.h`. The file name field (`dir_name`) contains the null terminated file name. The first byte is set to zero (a null string) to indicate a deleted or unused entry. The address field (`dir_fd_addr`) contains the LBN of the first file descriptor block.

```
#define MAXNAME    43                              /* size of name */
#define DIRENTSIZE 64                           /* size of directory entry */
typedef struct dirent {
     char     dir_name[MAXNAME+1],                 /* name of file */
              dir_unused[DIRENTSIZE-MAXNAME-sizeof(u_int32)-1];
     u_int32     dir_fd_addr;                      /* where file's FD is */
} dirent;
```

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 enables you to access any byte(s) or block(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 blocks.

A device is opened for physical I/O by appending the “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 block, perform a seek to the address computed by multiplying the LBN by the logical block size. For example, to read physical disk block 3 on media with a logical block size of 256, a seek is performed to address 768 (256*3), followed by a read system call requesting 256 bytes.

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

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 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 can open the raw device for write permit. Non-super users are only permitted to read the identification block (LBN 0). Attempts to read past this return an end-of-file error.

RECORD LOCKING

Record locking is a general term referring to preserving the integrity of files that more than one user or process can access. This involves recognizing when a process is trying to read a record another process may be modifying and deferring the read request until the record is safe. This process is referred to as conflict detection and prevention. RBF record locking also handles non-sharable files and deadlock detection.

OS-9 record locking is transparent to application programs. Most programs may be written without special concern for multi-user activity.

Record Locking and Unlocking

Conflict detection must determine when a record is being updated. RBF provides true record locking on a byte basis. A typical record update sequence is as follows:

```
_os_read(path, count, buffer)   program reads record;
                                RECORD IS LOCKED
 .
 .                              program updates record
 .
_os_seek(position)              reposition to record
_os_write(path, count, buffer)  record is rewritten;
                                RECORD IS RELEASED
```

When a file is opened in update mode, any read operation locks out the record because RBF is not aware if the record may be updated. The record remains locked until the next read, write, or close operation occurs. Reading files opened in read or execute modes does not lock the record because records cannot be updated in these modes.

A subtle problem exists for programs using a database and occasionally updating its data. When you look up a particular record, the record may be locked out indefinitely if the program neglects to release it. This problem is characteristic of record locking systems and can be avoided by careful programming.

Only one portion of a file may be locked out at one time. If an application requires more than one record to be locked out, multiple paths to the same file may be opened 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. Alternately, the entire file may be locked out, the records updated, and the file released.

Non-Sharable Files

You can lock files when an entire file is considered unsafe for use by more than one user. On rare occasions, it is necessary to create a nonsharable file. A non-sharable file can never be accessed by more than one process at a time.

To create a non-sharable file, set the exclusive access (`x`) bit in the file attribute byte. The bit can be set when the file is created, or later using the `attr` utility.

If the exclusive access bit has been set, only one process may open the file at a time. If another process attempts to open the file, an error (`EOS_SHARE`) is returned.

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

One example might be when a file is being sorted. If the file is opened as a non-sharable file, it is treated as though it had an exclusive access attribute. If the file has already been opened by another process, an error (`EOS_SHARE`) is returned.

A necessary quirk of non-sharable files is they may be duplicated using the I_DUP system call, or inherited. Therefore, a non-sharable file may actually become accessible to more than one process at a time. Non-sharable only means the file may be opened once. It is usually a 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. The end-of-file is kept locked until a read or write is performed that is not at end-of-file. EOF lock is the only case when a write call automatically locks out any of the file. This avoids problems that may otherwise occur when two users want to extend a file simultaneously.

An interesting and useful side effect occurs when a program creates a file for sequential output. As soon as the file is created, EOF lock is gained, and no other processes can *pass* the writer in processing the file.

For example, if an assembly listing is redirected to a disk file, a spooler utility may open and begin listing the file before the assembler writes 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 simultaneously attempt to gain control of the same two disk areas. If each process gets one area (locking out the other process), both processes can become stuck permanently, waiting for a segment that can never become free. This situation is a general problem not restricted to any particular record locking method or operating system.

If this occurs, a deadlock error (EOS_DEADLK) is returned to the process that detects the deadlock. The easiest way to avoid deadlock errors 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 data file before the index file.

When a deadlock error occurs, a program cannot simply retry the operation in error. If all processes used this strategy, none would ever succeed. At least one process must release control over a requested segment for any to proceed.

RECORD LOCKING DETAILS FOR I/O FUNCTIONS

Record locking details are described, by function, in the following subsections.

Open/Create

When opening files, the most important guideline to follow is not to open a file for update if it is only necessary to read. Files open for read only do not lock out records and generally help the system run faster. If shared files are routinely opened for update on a multi-user system, you may 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 only conforms to record lockouts with other users of the @ file.

Read/ReadLine

`Read` and `ReadLine` lock out 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 requested number of bytes.

For example, if a `ReadLine` call is made 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 also 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
- an `I_SETSTAT, SS_LOCK` set status call is issued

Releasing a record does not normally release EOF lock. A read or write of zero bytes releases any record lock, EOF lock, or file lock.

Write/WriteLine

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

Seek

Seek does not effect record locking.

SetStatus

Two SetStats have been included for the convenience of record locking:

`I_SETSTAT, SS_LOCK` Locks or releases part of a file.

`I_SETSTAT, SS_TICKS` Sets the length of time a program waits for a locked record.

See the `I_SETSTAT` entry in *Chapter 8, OS-9 System Calls* for a description of the codes.

FILE SECURITY

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

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

The attribute word has three sets of bits indicating whether a file may be opened for read, write, or execute by the owner, group, or public.

- An owner is a user with the same owner ID.
- The group includes all users with the same group ID.
- The public includes all users.

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 search a directory, you cannot read any files in that directory.

A 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 can not be loaded.

The RBF file descriptor stores the group/user ID in two 16-bit fields (`fd_group` and `fd_owner`).

⚠ The system manager must exercise caution when assigning group/user IDs.

PC FILE MANAGER (PCF)

While most of this chapter covers RBF issues, there are some PCF issues the user needs to know.

PCF is a reentrant subroutine package that handles I/O service requests for random-access PC-DOS/MS-DOS disk devices. PCF can handle any number of such devices simultaneously, and is responsible for maintaining the defined logical file structure on the PC-DOS/MS-DOS disk drive.

PCF supports FAT12, FAT16, and FAT32 file formats. Long file names (called VFAT), introduced with the advent of Windows 95, are fully supported. PCF will automatically choose the correct FAT algorithms for the device that is accessed. When creating a FAT file system, FAT12 should be used for devices under 32MB in size and FAT16 should be used for devices under 2GB in size. The requirements of FAT32 increase overhead and will slow down disk access.

Getting Top Performance from PCF

While PCF has been designed to achieve as much performance as possible, there are a few steps that applications can take to insure that PCF achieves maximum throughput:

- **Initialize all PCF devices**
  For performance reasons, PCF reads the entire disk's FAT into memory at open time. If the device is not initialized, the reading of the FAT can occur as many times as a file is opened on the device. To insure the FAT is read once per device, initialize the device before using it. This will decrease file open times, especially on slower devices such as floppy drives or large devices such as hard drives larger than 512MB.

- **Pre-extend files when writing**
  One way of increasing write performance is to pre-extend the file's size by using the `_os_ss_size()` function. Note that the `FAM_SIZE` bit in `_os_create()` is not recognized by PCF.

Differences from RBF

While PCF maintains very good compatibility with existing OS-9 disk utilities, there are some subtle differences that should be noted.

- Absence of Record Locking
  Unlike RBF, PCF does not employ record locking on a file. However, to prevent conflicts between processes, device locking is used at each entry point of the PCF file manager.
- `FAM_SIZE`
  Under RBF, a typical way to pre-extend the size of a file at create time is to pass `FAM_SIZE` as a parameter to the `_os_create()` function; however, the PCF file manager does not recognize this parameter. If file pre-sizing is desired, use the `_os_ss_size()` function.
- The PCF directory structure has a different format than that of RBF. If the application reads the directory raw and parses the entries, it must be written to accommodate the PCF directory format. It is highly recommended that an application which needs to read directory structure information use the portable functions: opendir(), readdir(), and closedir(). These functions are compatible with all OS-9 file storage managers.

7 Resource Locking

This chapter describes the lock structure definition, lock creation, signal lock relationships, and FIFO buffer usage. It includes the following topics:

- Overview
- Preallocate Locks as Part of the Resource
- Signals and Locks
- FIFO Buffers

OVERVIEW

The OS-9 I/O system uses resource locking calls to provide exclusive access to critical regions and help ensure proper resource management. If you write file managers or drivers, review this chapter for an explanation of resource locking and implementation details.

Resource locking helps prevent data corruption by limiting process access to critical sections of code; it protects data structures from simultaneous modification by multiple processes. To manage processes waiting to enter critical areas, resource locking provides an associated queue. The queue orders lock requests according to the relative priority of the calling process.

Resource locking is only available in system state.

The following are the OS-9 resource locking calls. Refer to *Chapter 8, OS-9 System Calls* for a detailed description of each call.

Table 7-1. OS-9 Resource Locking Calls

Call	Description
`F_ACQLK`	Acquire ownership of a resource lock.
`F_CAQLK`	Conditionally acquire ownership of a resource lock.
`F_CRLK`	Create a new resource lock descriptor.
`F_DELLK`	Delete an existing lock descriptor.
`F_RELLK`	Release ownership of a resource lock.
`F_WAITLK`	Activate the next process waiting to acquire a lock, and suspend the current process.

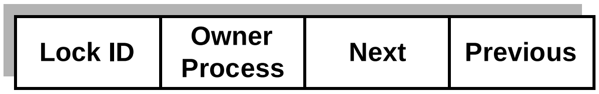
Lock Structure Definition

The lock structure definition for the kernel is as follows:

```
typedef struct lock_desc *lock_id;
typedef struct lock_desc {
     lock_id     l_id;        /* lock identifier */
     Pr_desc     l_owner,     /* current owner */
                 l_lockqn,    /* next process in lock list */
                 l_lockqp;    /* previous process in lock list */
} lk_desc, *Lk_desc;
```

Conceptually, this structure could be shown as:

Figure 7-1. Lock Structure



Lock ID	Owner Process	Next	Previous

The next and previous boxes represent the queuing capabilities of resource locking calls. When one or more processes are waiting to acquire a lock, they work with corresponding process descriptor fields to determine which process should receive the lock next. Lock requests are queued in the order in which they are received, according to their relative priority. Higher priority processes are queued ahead of lower priority processes.

Create and Delete Resource Locks

OS-9 provides a call to dynamically create and initialize a resource lock. The `F_CRLK` call allocates data space for the lock, initializes the associated queue, and sets the lock ownership to a free state. A lock identifier is returned for subsequent use by the lock calls.

The lock identifier is the address of the lock structure.

When a lock is no longer needed, you can use the `F_DELLK` call to deallocate it. The data space for the lock is returned to the system. Prior to deleting a lock you must ensure any processes waiting in its queue are removed from the queue and re-activated. The `F_DELLK` call does not check the queue for waiting processes; it is the responsibility of the application to empty the waiting queue of the lock.

The following C language example demonstrates how to dynamically create and delete a resource lock.

```
#include <types.h>
#include <lock.h>

Lk_desc lock;       /* declare a pointer to a lock structure */

                    /* dynamically allocate a new lock */
if ((error = _os_crlk(&lock)) != SUCCESS)
      return error;

/* an example use of the lock */
if ((error = _os_acqlk(lock, &signal)) != SUCCESS)
      return error;
/* delete the lock */
_os_dellk (lock);
```

PREALLOCATE LOCKS AS PART OF THE RESOURCE

To reduce the overhead and memory fragmentation caused by dynamically created locks, you can declare the lock structure for a given resource as part of the resource structure. Prior to using the lock, you must initialize the lock structure fields.

For example:

```
#include <types.h>
#include <const.h>
#include <lock.h>
#include <process.h>

/* Resource declaration with the lock structure included */
struct xyz {
      lk_desc lock;
      int a;
      char *b;
      unsigned c;
} resource;

/* set the lock identifier */
resource.lock.l_id = &resource.lock;

/* declare the lock free */
resource.lock.l_owner = NULL;

/* initialize the lock structure's queue pointers */
resource.lock.l_lockqp = resource.lock.l_lockqn =
      FAKEHD(Pr_desc, resource.lock.l_lockqn, p_lockqn);
```

The `FAKEHD` initialization macro is located in the `const.h` header file.

At this point, the lock within the resource structure is ready for use. Subsequent lock calls are made by passing the address of the lock as its identifier. The following acquire lock example demonstrates this:

```
/* use a lock declared within a resource structure */
if ((error = _os_acqlk(&resource.lock, &signal)) != SUCCESS)
          return error;
```

SIGNALS AND LOCKS

Locks have an associated queue used for suspending processes waiting to acquire a busy lock. If the lock is busy, the acquiring process is placed in the queue according to the relative priorities of any other waiting processes. When the owner process releases its ownership of the lock, the next process in the queue is activated and granted sole ownership of the lock. On the new owner's next time slice, the process returns from the acquire lock system call without error and continues to execute from that point. Normally, this is the proper sequence of events; the active process has ownership of the resource. But it is possible for a process to be prematurely activated prior to acquiring ownership of the lock.

If, for example, the process receives a signal while waiting in the lock queue, the process is activated without acquiring the lock and the acquire lock call returns an EOS_SIGNAL error. To avoid this error, it is critical that applications check the return value of the acquire lock calls to validate whether or not the active process has gained ownership of the lock. If a process is activated by a signal, the application writer determines how to respond to the error condition. The application may abort its operation and return with an error, or ignore the signal and attempt to re-acquire the lock. Depending on the application, either action may be appropriate.

Signal Sensitive Locks

The following example uses a lock to protect a critical section of code modifying a non-sharable resource. This example is completely sensitive to any signals a process may receive while waiting to acquire the lock. A process receiving a signal while waiting in this lock's queue is activated and the acquire lock call returns the error EOS_SIGNAL.

```
#include <lock.h>
#include <types.h>
#include <errno.h>

lk_desc lock;
signal_code signal;

/* acquire exclusive access to the resource */
if ((error = _os_acqlk(&lock, &signal)) != SUCCESS)
     return error;

<critical section>

/* release exclusive access to the resource and activate the next process */
_os_rellk(&lock);
```

Ignoring Signals

There may be situations when a process is prematurely activated by a signal, and it is not appropriate for the application to simply return an error. In this case, the application may ignore the activating signal and error and attempt to re-acquire the lock.

The activating signal is not lost. The operating system queues it on behalf of the process. Upon return from system state, the signal is delivered to the process through its signal intercept routine.

This acquire lock example demonstrates how to use locks that ignore signals.

```
#include <lock.h>
#include <types.h>
#include <errno.h>

lk_desc lock;
signal_code signal;
while ((error = _os_aqclk(&lock, &signal)) != SUCCESS) {
     if (error == EOS_SIGNAL)
          continue;            /* signal received, ignore it */
     else
     return error;             /* some other erroneous condition */

      <critical section>

     /* release exclusive access to the resource and activate the next
process */
      _os_rellk(&lock);
}
```

The following is an example of a lock that is partially sensitive to signals. It ignores any non-deadly signals a process might receive, but returns an error for any deadly signal. In this case, a deadly signal is any signal with a value less than 32.

```
#include <lock.h>
#include <types.h>
#include <errno.h>

lk_desc lock;
signal_code signal;
while ((error = _os_aqclk(&lock, &signal)) != SUCCESS) {
      if (error == EOS_SIGNAL) {
            if (signal >= 32)
            continue;           /* signal greater than 32 received,
ignore it */
            else
                  return error; /* signal less than 32 received */
      }
      else break;             /* some other erroneous condition */
       <critical section>
       /* release exclusive access to the resource and activate the
next process */
      _os_rellk(&lock);
}
```

FIFO BUFFERS

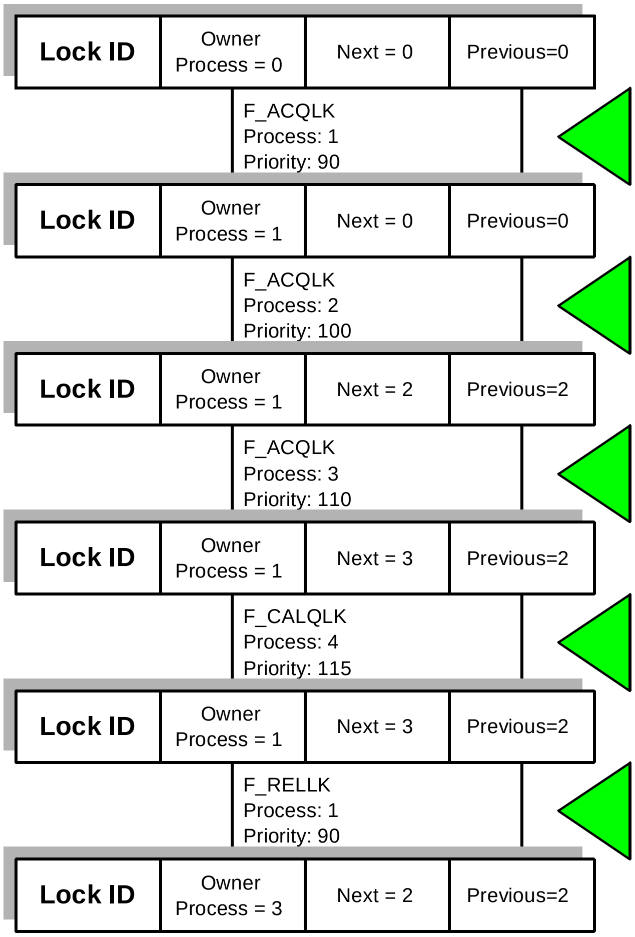
You can use locks to synchronize the reader and writer of a FIFO buffer resource. The resource has an associated lock; any reader or writer requiring access to the resource must first acquire the resource lock. After acquiring the resource, the process may proceed to modify the buffer. If during the course of modification the reader empties the buffer or the writer fills the buffer, the F_WAITLK call suspends the process to wait for more data to enter or leave the buffer.

```
#include <lock.h>
#include <types.h>
#include <errno.h>

lk_desc lock;
signal_code signal;

/* acquire exclusive access to the resource */
if ((error = _os_acqlk(&lock, &signal)) != SUCCESS) return error;

/* loop until total number of bytes is read/written */
while (bytes_read/bytes_written < bytes_to_read/bytes_to_write) {

       /* check for bytes available to read/write */
      if (bytes_available == 0) {

       /* no bytes available, so release the ownership of the lock, */
        /* activate the reader/writer if it is waiting, and
unconditionally */
       /* suspend the current reader/writer                        */
        if ((error = _os_waitlk(&lock, &signal)) != SUCCESS)
                 return error;
      }
      else {

      <transfer bytes>

     }
}
/* number of bytes to read/write has been satisfied, so release lock
*/
_os_rellk(&lock);
```

Process Queuing

The diagram below is a conceptual illustration of the queuing process and the effect of various calls on the lock structure.

Figure 7-2. Effect of Various Calls on the Lock Structure



The owner process = 0, so the lock is available to process 1. Process 1 now owns the lock.

This call places process 2 in the queue. Process 2 must wait until process 1 releases the lock before it can be the owner process.

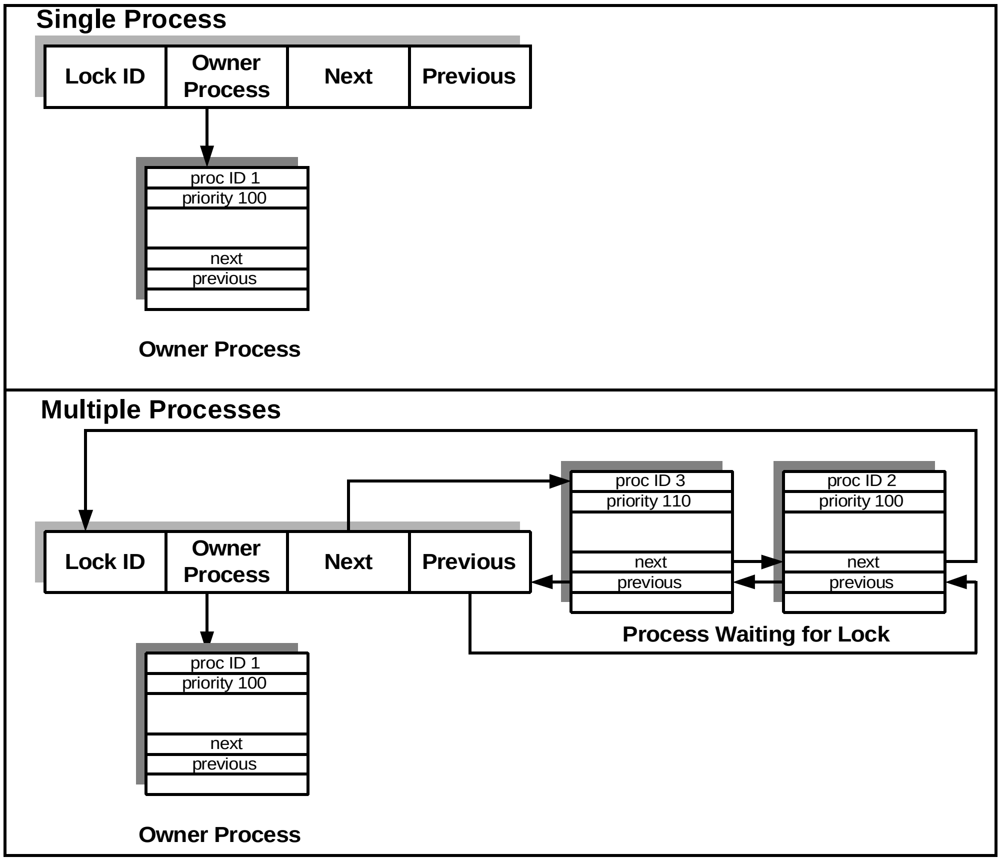
This call causes the queue to be re-ordered because process 3 has a higher priority than process 2. Process 3 will be the next one to acquire the lock.

This conditional acquire call has no effect on the lock structure; it is only performed if the lock is now owned by another process. In this case, it returns error EOS_NOLOCK .

This call releases process 1. The lock is now available to process 3. Process 2 moves up in line; it can acquire the lock after process 3 is released.

The following figure show the locking sequence with one process and with multiple processes.

Figure 7-3. Locking Sequence



8 OS-9 System Calls

This chapter explains how to use OS-9 system calls and contains an alphabetized list of all OS-9 system calls. It includes the following topics:

- Using OS-9 System Calls
- System Calls Reference

USING OS-9 SYSTEM CALLS

System calls are used to communicate between the OS-9 operating system and C or assembly language programs. There are four general categories of system calls:

- User-state system calls
- I/O system calls
- System-state system calls
- System-state I/O system calls

All of the OS-9 system calls require a single parameter to be passed to the operating system, called the parameter block. Parameter blocks are the means by which applications and system software pass parameters to the operating system for service requests. When a system call is performed, a pointer to the associated service request parameter block is passed to the operating system. The operating system acquires the specific parameters it needs for the service request from the parameter block and returns any result parameters through the parameter block.

Every system call parameter block contains the same substructure, `syscb`. `syscb` contains:

- An identifier of the service request
- The edition number of the service request interface
- The size of the associated parameter block
- A result field for returning error status

For programming convenience, a C language system call library containing a C interface for each of the OS-9 system calls is provided. A complete description of the C language interface for each of the system calls can be found in the *Ultra C Library Reference*.

_oscall Function

There is a single routine located in the system call library providing the gateway into the operating system. The `_oscall` function expects a parameter block pointer and uses whatever trap or software interrupt facility is available on a given hardware platform to enter into the operating system.

The `_oscall()` request is a common interface to the kernel and the mechanism by which all OS-9 system calls are made. `_oscall()` has one parameter: the address of a parameter block or structure belonging to the system call. Each OS-9 system call binding creates a parameter block that is passed to the kernel by `_oscall()`.

For example, the C binding for the `F_FMOD` system call fills the parameter block and passes the address of the block directly to the kernel through `_oscall()`:

```
#include "defsfile"

/* _os_fmod - find module directory entry service request. */
_os_fmod(type_lang, moddir_entry, mod_name)
u_int16   *type_lang;
Mod_dir   *moddir_entry;
u_char    *mod_name;
{
   register error_code error;
   f_findmod_pb pb;          /* declare parameter block of appropriate
type */

   pb.cb.code = F_FMOD;                        /* fill parameter block
field;
                                                   fn code defined in
funcs.h */
   pb.cb.param_size = sizeof f_findmod_pb;     /* fill parameter block
field */
   pb.cb.edition = _OS_EDITION;                 /* fill edition number
*/

   pb.type_lang = *type_lang;                  /* fill parameter block
field */
   pb.mod_name = mod_name;                     /* fill parameter block
field */
    if ((error = _oscall(&pb)) == SUCCESS)  { /* make _oscall */
        *type_lang = pb.type_lang;              /* return value */
        *moddir_entry = pb.moddir_entry;        /* return value */
   }
    return error;
}
```

For more information about installing system calls, refer to the description of the `F_SSVC`.

A complete list of structures for OS-9 system calls is included in *Chapter 1, System Overview*.

Using the System Calls

The typical sequence for executing an OS-9 system call would be as follows:

1. Allocate a parameter block specific to the system call.
2. Initialize the parameter block including the system sub-block.
3. Call the operating system (through `_oscall`).
4. Check for errors upon return.
5. Process return parameters, if applicable.

All of the predefined parameter blocks for the OS-9 are located in the `srvcb.h` header file. Each system call description within this chapter includes a full description of the parameter block structure specific to the system call, as well as a full summary of the functionality of the system call.

System Call Descriptions

The OS-9 Attributes field indicates the state of each call, whether the call is an I/O call, and if the call can be used during an interrupt. The characteristic for each field (for example user, system, I/O, or interrupt) is listed where appropriate. In addition, the OS-9 Attributes table indicates whether a function is thread-safe or -unsafe.

System-state system calls are privileged. They may be executed only while OS-9 is in system state (for example, when it is processing another service request or executing a file manager or device driver). System-state functions are included in this manual primarily for the benefit of those programmers who write device drivers and other system-level applications.

Some system calls generate errors themselves; these are listed as Possible Errors. If the returned error code does not match any of the given possible errors, it was probably returned by another system call made by the main call. In the system call description section, strings passed as parameters are terminated by a null byte.

If you use the system calls from assembly language, do not alter registers.

Interrupt Context

If you use any system calls in an interrupt service routine that are not listed in the following table, you can corrupt the integrity of your system.

F_ALARM, A_RESET	F_EVENT, EV_SET	F_GPRDBT	F_SUSER
F_APROC	F_EVENT, EV_SETAND	F_ICPT	F_SYSID
F_CAQLK	F_EVENT, EV_SETOR	F_ID	F_TIME
F_CCTL (System State)	F_EVENT, EV_SETR	F_INITDATA	F_UACCT
F_CLRSIGS	F_EVENT, EV_SETXOR	F_MOVE	I_CIOPROC
F_CPYMEM	F_EVENT, EV_SIGNL	F_SEND	I_GETDL
F_EVENT, EV_INFO	F_EVENT, EV_UNLNK	F_SETSYS	I_GETPD
F_EVENT, EV_LINK	F_EVENT, EV_WAIT	F_SPRIOR	I_GETSTAT, SS_COPYPD
F_EVENT, EV_PULSE	F_EVENT, EV_WAITR	F_SSPD	I_GETSTAT, SS_DEVNAME
F_EVENT, EV_READ	F_FMOD	F_SSVC	I_GETSTAT, SS_DEVTYPE

SYSTEM CALLS REFERENCE

The following section describes the system calls in detail.

F_ABORT

Emulate Exception Occurrence

Headers

```
#include <regs.h>
```

Parameter Block Structure

```
typedef struct f_abort_pb {
   syscbcb;
   u_int32strap_code,
      address,
      except_id;
} f_abort_pb, *F_abort_pb;
```

Description

`F_ABORT` emulates the occurrence of an exception. This service request executes the same recovery code in the OS used to recover from exceptions occurring in the system. The OS responds to this service just as it would if the specified exception had actually occurred. This allows applications or system extension modules to force an exception condition without actually triggering the exception. An application may use this service to test its exception handlers that were installed using the `F_STRAP` service.

This service is used by some of the floating-point emulation extension modules on processors lacking hardware floating-point support to trigger floating-point exception conditions detected during software emulation of floating-point instructions. The service emulates the floating-point exceptions that would have occurred if the floating-point instructions had been executed by real hardware.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`strap_code` is the associated code used in the `F_STRAP` service request to setup an exception handler. It is the `F_STRAP` code of the exception to emulate. The `F_STRAP` codes are defined in the `reg<CPU>.h` header file for the target CPU platform.

`address` is the address of where the exception is to have occurred.

`except_id` is the hardware vector identifier of the exception to emulate. The exception vector identifiers are defined in the `reg<CPU>.h` header file for the target CPU platform.

See Also

F_STRAP

F_ACQLK

Acquire Ownership of Resource Lock

Headers

```
#include <lock.h>
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_acqlk_pb {
   syscb cb;
   lock_id lid;
   signal_code signal;
} f_acqlk_pb, *F_acqlk_pb;
```

Description

`F_ACQLK` acquires ownership of a resource lock (it attempts to gain exclusive access to a resource).

If the lock is not owned by another process, the calling process is granted ownership and the call returns without error.

If the lock is already owned, the calling process is suspended and inserted into a waiting queue for the resource based on relative scheduling priority.

When ownership of the lock is released, the next process in the queue is granted ownership and is activated. The activated process returns from the system call without error. If, during the course of waiting on a lock, a process receives a signal, the process is activated without gaining ownership of the lock. The process returns from the system call with an `EOS_SIGNAL` error code and the signal code returned in the `signal` pointer.

If a waiting process receives an `S_WAKEUP` signal, the signal code does not register and will be zero.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`lid` is the lock identifier of the lock you are attempting to acquire.

`signal` is the signal prematurely terminating the acquisition of the lock.

Possible Errors

`EOS_SIGNAL`

See Also

`F_CAQLK`
`F_CRLK`
`F_DELLK`
`F_RELLK`
`F_WAITLK`

Refer to Chapter 6 for more information on locks.

F_ALARM (System-State)

System-State OS-9 Alarm Request

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_salarm_pb {
   syscb cb;
   alarm_id alrm_id;
   u_int16 alrm_code;
   u_int32 time,
      flags;
   u_int32 (*function)();
   void *func_pb;
} f_salarm_pb, *F_salarm_pb;
```

Description

The system-state alarm requests execute a system-state subroutine at a specified time. They are provided for functions such as turning off a disk drive motor if the disk is not accessed for a period of time.

System-state alarms, as well as user-state alarms, always belong to some process. This process, for system-state alarms, is either the creating process (if the `TH_SPOWN` bit was `0` when the process had the operating system create the alarm) or the system process (if the `TH_SPOWN` bit was `1` when the process had the operating system create the alarm). For user-state alarms, they always belong to the creating process and never the system process. If a process gives ownership of an alarm to the system process, then the alarm remains in the system until either it expires, or some system-state process deletes it. In all other respects, system-state alarms behave as user-state alarms.

The time interval is the number of system clock ticks (or 1/256 second) to wait before an alarm signal is sent. If the high order bit is set, the low 31 bits are interpreted as 1/256 second. All times are rounded up to the nearest tick.

The alarm functions do not return any error code if the operating system cannot wait for the requested time due to an overflow when converting a time from 256ths-of-a-second into clock ticks. This only occurs if you specify a time in 256ths-of-a-second and the system clock ticks occur at a rate greater than 512 ticks-per-second. If an overflow occurs, the operating system waits for the longest delay possible.

The following system-state alarm functions are supported:

Alarm	Description
`F_ALARM, A_ATIME`	Executes a subroutine at a specified time.
`F_ALARM, A_CYCLE`	Executes a subroutine at specified time intervals.
`F_ALARM, A_DELET`	Removes a pending alarm request.
`F_ALARM, A_RESET`	Resets an existing alarm request.
`F_ALARM, A_SET`	Executes a subroutine after a specified time interval.

During an `A_RESET` request, the `TH_SPOWN` bit has the following meaning: if `0`, allow the calling process to update only its own alarms; if `1`, allow the calling process to update any alarm.

During an `A_DELETE` request, the `TH_SPOWN` bit has the following meaning: if `0`, allow the calling process to delete only its own alarms; if `1`, allow the calling process to delete any alarm. If the `alarm_id` field is `0` and the `TH_SPOWN` bit is `1`, the operating system deletes all alarms belonging to the system process.

System-state alarms are run by the system process. They should not perform any function resulting in any kind of queuing, such as F_SLEEP; F_WAIT; F_LOAD; and F_EVENT, EV_WAIT. When such functions are required, the caller must provide a separate process to perform the function, rather than an alarm.

IRQ routines cannot create or delete alarms since such actions cause memory allocations/deallocations, that are illegal from an IRQ routine. The way to handle such things is to create the alarms before the IRQ routine needs them, and then have the IRQ routine use only RESETs, that are legal in IRQ routines.

For non-system, process-owned alarms, the user number in the system process descriptor changes temporarily to the user number of the original process.

If an alarm execution routine suffers any kind of bus trap, address trap, or other hardware-related error, the system crashes.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`alrm_id` is the alarm identifier returned by the system call. The alarm ID may subsequently be used to delete the alarm, if desired, by using the F_ALARM, A_DELET alarm call.

`alrm_code` is the particular alarm function to perform.

`time` is the specified time.

`flags` is one of the following two alarm flags defined in `<process.h>`:

Flag	Value	Description
`TH_DELPB`	`0x00000001`	Indicates the associated function parameter block's memory should be returned to the system after executing the alarm function.
`TH_SPOWN`	`0x00000002`	Indicates the system-state alarm should be owned by the system process and not the current process.

`function` is the function to be executed.

`func_pb` points to the function's parameters block.

Possible Errors

```
EOS_NOCLK
EOS_NORAM
EOS_PARAM
EOS_UNKSVC
```

See Also

F_ALARM (User-State)
F_EVENT, EV_WAIT
F_LOAD
F_SLEEP
F_WAIT

F_ALARM (User-State)

User-State Set Alarm Clock

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_alarm_pb {
  syscb cb;
  alarm_id alrm_id;
  u_int16 alrm_code;
  u_int32 time;
  signal_code signal;
} f_alarm_pb, *F_alarm_pb;
```

Description

The user-state alarm requests enable a user process to create 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 1/256 second) to wait before an alarm signal is sent. If the high order bit is set, the low 31 bits are interpreted as 1/256 second.=

All times are rounded up to the nearest system clock tick.

The alarm functions do not return any error code if the operating system cannot wait for the requested time due to an overflow when converting a time from 256ths-of-a-second into clock ticks. This only occurs if you specify a time in 256ths-of-a-second and the system clock ticks occur at a rate greater than 512 ticks-per-second. If an overflow occurs, the operating system waits for the longest delay possible.

The following user-state alarm functions are supported:

Function	Description
F_ALARM, A_ATIME	Send signal at specified time.
F_ALARM, A_CYCLE	Send signal at specified time intervals.
F_ALARM, A_DELET	Remove pending alarm request.
F_ALARM, A_RESET	Reset existing alarm request to occur at a newly specified time.
F_ALARM, A_SET	Send signal after specified time interval.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

`cb` is the control block header.

`alrm_id` is the alarm identifier returned by the system call. The alarm ID may subsequently be used to delete the alarm, if desired, by using the `F_ALARM`, `A_DELET` alarm call.

`alrm_code` is the particular alarm function to perform.

`time` is the specified time.

`signal` is the signal value originally belonging to the alarm.

Possible Errors

EOS_BPADDR
EOS_NORAM
EOS_PARAM
EOS_UNKSVC

See Also

F_ALARM (System-State)

F_ALARM, A_ATIME

Send Signal At Specified Time (User State)
Execute Subroutine At Specified Time (System State)

Headers

```
#include <types.h>
```

Parameter Block Structure

If OS-9 is in system state, see `F_ALARM (System-State)` for the parameter block structure. Otherwise, see `F_ALARM (User-State)` for the parameter block structure.

Description

`A_ATIME` sends one signal at the specified time in user state or executes a subroutine at the specified time in system state.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`alrm_id` is the alarm identifier returned by the system call. The alarm ID may subsequently be used to delete the alarm, if desired, by using the `F_ALARM`, `A_DELET` alarm call.

`signal` is the signal code of the signal to send.

`time` is the specified time. The value is considered to be an absolute value in seconds since 1 January 1970 Greenwich Mean Time.

Possible Errors

`EOS_NOCLK`
`EOS_NORAM`
`EOS_PARAM`

See Also

F_ALARM, A_SET
F_ALARM (System-State)
F_ALARM (User-State)

F_ALARM, A_CYCLE

Send Signal at Specified Time Intervals

Headers

```
#include <types.h>
```

Parameter Block Structure

If OS-9 is in system state, see F_ALARM (System-State) for the parameter block structure. Otherwise, see F_ALARM (User-State) for the parameter block structure.

Description

A_CYCLE sends a signal after the specified time interval has elapsed and then resets the alarm. This provides a recurring periodic signal.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`alrm_id` is the returned alarm ID.

`alrm_code` is the particular alarm function to perform (in this case, A_CYCLE).

`signal` is the signal code of the signal to send.

`time` specifies the time interval between signals. The time interval may be specified in system clock ticks; or if the high order bit is set, the low 31 bits are considered a time in 1/256 second. The minimum time interval allowed is two system clock ticks.

Possible Errors

EOS_NOCLK
EOS_NORAM
EOS_PARAM

See Also

F_ALARM, A_SET
F_ALARM (System-State)
F_ALARM (User-State)

F_ALARM, A_DELET

Remove Pending Alarm Request

Headers

```
#include <types.h>
```

Parameter Block Structure

If OS-9 is in system state, see `F_ALARM (System-State)` for the parameter block structure. Otherwise, see `F_ALARM (User-State)` for the parameter block structure.

Description

`A_DELET` removes a cyclic alarm, or any alarm that has not expired.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`alrm_id`	specifies the alarm identification number. If `alrm_id` is zero, all pending alarm requests are removed.

Possible Errors

```
EOS_BPADDR
EOS_IBA
EOS_NORAM
EOS_PARAM
```

See Also

`F_ALARM, A_SET`
`F_ALARM (System-State)`
`F_ALARM (User-State)`

F_ALARM, A_RESET

Reset Existing Alarm Request

Headers

```
#include <types.h>
```

Parameter Block Structure

If OS-9 is in system state, see `F_ALARM (System-State)` for the parameter block structure. Otherwise, see `F_ALARM (User-State)` for the parameter block structure.

Description

`A_RESET` resets an existing alarm to occur at the newly specified time. It does not reset any other characteristics of the original alarm.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`alrm_id` is the ID of the alarm to reset.

`signal` is the signal code of the signal to send.

`time` may be specified in system clock ticks; or if the high order bit is set, the low 31 bits are considered a time in 1/256 second. The minimum time interval allowed is two clock ticks.

Possible Errors

```
EOS_NOCLK
EOS_NORAM
EOS_PARAM
```

See Also

F_ALARM, A_SET
F_ALARM (System-State)
F_ALARM (User-State)

F_ALARM, A_SET

Send Signal After Specified Time Interval

Headers

```
#include <types.h>
```

Parameter Block Structure

If OS-9 is in system state, see `F_ALARM (System-State)` for the parameter block structure. Otherwise, see `F_ALARM (User-State)` for the parameter block structure.

Description

`A_SET` sends one signal after the specified time interval has elapsed.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`alrm_id` is the alarm identifier returned by the system call. The alarm ID can subsequently be used to delete the alarm, if desired, by using the `A_DELET` alarm call.

`signal` is the signal code of the signal to send.

`time` can be specified in system clock ticks; or if the high order bit is set, the low 31 bits are considered a time in 1/256 second. The minimum time interval allowed is two system clock ticks.

Possible Errors

```
EOS_BPADDR
EOS_NORAM
EOS_PARAM
```

See Also

`F_ALARM, A_DELET`
`F_ALARM (System-State)`
`F_ALARM (User-State)`

F_ALLPRC

Allocate Process Descriptor

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_allprc_pb {
    syscb cb;
    process_id proc_id;
    Pr_desc proc_desc;
}  f_allprc_pb, *F_allprc_pb;
```

Description

F_ALLPRC allocates and initializes a process descriptor. The address of the descriptor is stored in the process descriptor table. Initialization consists of clearing the descriptor and setting its process identifier.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	is a returned value. It is the process ID of the new process descriptor.
proc_desc	is a returned value. It points to the new process descriptor.

Possible Errors

EOS_PRCFUL

F_ALLTSK

Allocate Task

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_alltsk_pb{
   syscb cb;
   Pr_desc proc_desc;
}  f_alltsk_pb, *F_alltsk_pb;
```

Description

F_ALLTSK is called just before a process becomes active to ensure the protection hardware is ready for the process. F_ALLTSK sets the protection hardware to the map for the process pointed to by proc_desc.

F_ALLTSK is only supported on systems with a memory protection unit (for example, all 80x86). The SSM module *must* be present in the bootfile.

If the SSM module is not present in the system, an EOS_UNKSVC error is returned. You should ignore this error.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb	is the control block header.
proc_desc	points to the process descriptor.

Possible Errors

EOS_UNKSVC

See Also

F_DELTSK

F_ALTMDIR

Set Alternate Working Module Directory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_altmdir_pb {
   syscb cb;
   u_char *name;
} f_altmdir_pb, *F_altmdir_pb;
```

Description

F_ALTMDIR establishes an alternate working module directory for a process.

When a process performs an F_LINK or F_FORK system call, the search for the specified target module begins in the process' current module directory. If that search fails, the alternate module directory is searched. This enables processes to link to or execute modules from different locations within system memory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

name
points to the name of the alternate working module directory.

Possible Errors

```
EOS_MNF
EOS_PERMIT
```

See Also

F_CHMDIR
F_FORK
F_LINK

F_APROC

Insert Process in Active Process Queue

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_aproc_pb {
    syscb cb;
    process_id proc_id;
}  f_aproc_pb, *F_aproc_pb;
```

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 new process is set to its priority, and the process is 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.

OS-9 does not preempt a process in system state (for example, the middle of a system call). However, OS-9 does set a bit (`TIMOUT` in `p_state`) in the process descriptor causing the process to surrender its time slice when it re-enters user state.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`

is the control block header.

`proc_id`

specifies the ID of the process to place in the active process queue.

Possible Errors

```
EOS_IPRCID
EOS_PERMIT
```

See Also

`F_NPROC`

F_CAQLK

Conditionally Acquire Ownership of Resource Lock

Headers

```
#include <lock.h>
```

Parameter Block Structure

```
typedef struct f_caqlk_pb {
   syscb cb;
   lock_id lid;
} f_caqlk_pb, *F_caqlk_pb;
```

Description

F_CAQLK conditionally acquires ownership of a resource lock.

If the lock is not owned by another process, the calling process is granted ownership and the call returns without error.

If the lock is already owned, the calling process is not suspended. Instead, it returns from the F_CAQLK call with an EOS_NOLOCK error and is not granted ownership of the resource lock.

Refer to Chapter 6 for more information on locks.

Attributes

Operating System:	OS-9
State:	System and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
lid	is the identifier of the lock you are attempting to acquire.

Possible Errors

EOS_NOLOCK

See Also

F_ACQLK
F_CRLK
F_DELLK
F_RELLK
F_WAITLK

F_CCTL (User-State)

User-State Cache Control

Headers

```
#include <types.h>
#include <cache.h>
```

Parameter Block Structure

```
typedef struct f_cache_pb {
   syscb cb;
   u_int32 control;
   void *addr;
   u_int32 size;
} f_cache_pb, *F_cache_pb;
```

Description

F_CCTL performs operations on the system instruction and/or data caches, if there are any.

If the C_ADDR bit of the control parameter is set, then the addr and size parameters are used to flush the specific target address from the cache. This functionality is only supported on hardware platforms with this capability.

Only system-state processes and super-group processes can perform the other precise operations of F_CCTL.

Any program that builds or changes executable code in memory should flush the instruction cache with F_CCTL before executing the new code. This is necessary because the hardware instruction cache may not be updated by data (write) accesses on certain hardware set ups and may therefore contain the unchanged instruction(s). For example, if a subroutine builds a system call on its stack, it should first use the F_CCTL system to flush the instruction cache before it executes the temporary instructions.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

cb
is the control block header.

control
specifies the cache operation.If control is zero, the system instruction and data caches are flushed. Only super-group processes can perform this operation. Only three bits may be used:

Bit	Name	Description
Bit 2	C_FLDATA	Flush data cache
Bit 6	C_FLINST	Flush instruction cache
Bit 8	C_ADDR	Indicates a specific target address for flush operation

addr
specifies the target address for the flush operation.

size
indicates the size of the target memory area to be flushed.

Possible Errors

EOS_PARAM

F_CCTL (System State)

System-State Cache Control

Headers

```
#include <types.h>
#include <cache.h>
```

Parameter Block Structure

```
typedef struct f_scache_pb {
   syscb cb;
   u_int32 control;
   u_int32 (*cctl)();
   void *cctl_data;
   void *addr;
   u_int32 size;
} f_scache_pb, *F_scache_pb;
```

Description

`F_CCTL` performs operations on the system instruction and/or data caches, if there are any.

Any program that builds or changes executable code in memory should flush the instruction cache by `F_CCTL` prior to executing the new code. This is necessary because the hardware instruction cache is not updated by data (write) accesses and may contain the unchanged instruction(s). For example, if a subroutine builds a system call on its stack, the `F_CCTL` system call to flush the instruction cache must be executed prior to executing the temporary instructions.

If the `C_GETCCTL` bit of `control` is set, `F_CCTL` returns a pointer to the cache control routine in the *cache* extension module and a pointer to that routine's static global data. This enables drivers and file managers to call the cache routine directly, rather than making a possibly time-consuming `F_CCTL` request.

The OS-9 kernel calls the cache extension module directly.

Attributes

Operating System:	OS-9
State:	System and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`control` specifies the cache operation. If `control` is zero, the system instruction and data caches are flushed. The following bits are defined in the `control` parameter for precise operation:

Bit	Name	Description
Bit 0	`C_ENDATA`	If set, enables data cache.
Bit 1	`C_DISDATA`	If set, disables data cache.
Bit 2	`C_FLDATA`	If set, flushes data cache.
Bit 3	C_INVDATA	If set, invalidates data cache.
Bit 4	`C_ENINST`	If set, enables instruction cache.
Bit 5	`C_DISINST`	If set, disables instruction cache.
Bit 6	`C_FLINST`	If set, flushes instruction cache.
Bit 7	C_INVINST	If set, invalidates instruction cache.
Bit 8	C_ADDR	Flags a target address for flush operation.
Bits 9-14		Reserved for future use by Microware.
Bit 15	`C_GETCCTL`	If set, returns a pointer to the cache control routine and cache static global data.

Bit	Name	Description
Bit 16	C_STODATA	If set, stores data cache (if supported by hardware).
Bits 17-31		Reserved for future use by Microware.

All other bits are reserved. If any reserved bit is set, an `EOS_PARAM` error is returned. *Precise operation* of `F_CCTL` can only be performed by system-state processes and super-group processes.

If the `C_ADDR` bit of the control parameter is set, then the `addr` and `size` parameters are used to flush the specific target address from the cache. This functionality is only supported on hardware platforms with this capability.

`cctl` is the returned cache routine.

`cctl_data` is the returned cache routine's static data.

`addr` specifies the target address for the flush operation.

`size` indicates the size of the target memory area to be flushed.

Possible Errors

`EOS_PARAM`

F_CHAIN

Load and Execute New Primary Module

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_chain_pb {
   syscb cb;
   u_int16 priority,
      path_cnt;
   u_char *mod_name,
      *params;
   u_int32 mem_size,
      param_size;
   u_int16 type_lang;
} f_chain_pb, *F_chain_pb;
```

Description

F_CHAIN executes a new program without the overhead of creating a new process. It is functionally similar to a F_FORK command followed by an F_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.

F_CHAIN executes as follows:

1. The process' old primary module is unlinked.
2. The system parses the name string of the new process' primary module (the program that is executed). Next, the current and alternate module directories are searched to see if a module with the same name and type/language is already in memory. If so, the module is linked. If not, the name string is used as the pathlist of a file to be loaded into memory. The first module in this file is linked.
3. The data memory area is reconfigured to the size specified in the new primary module's header.
4. Intercepts and pending signals are erased.

5. The following structure definition is passed to the initial function of the new module (this is identical to `F_FORK`).

6. typedef struct {

```
process_id   proc_id;     /* process ID */
owner_id     owner;       /* group/user ID */
prior_level  priority;    /* priority */
u_int16      path_count;  /* of I/O paths inherited*/
u_int32      param_size,  /* size of parameters */
               mem_size;    /*total initial memory
allocated*/
  u_char       *params,     /* parameter pointer */
               *mem_end;    /* top of memory pointer */
 Mh_com        mod_head;    /*primary (forked) module
ptr*/
} fork_params, *Fork_params;
```

The minimum overall data area size is 256 bytes.

`F_CHAIN` never returns to the calling process. If an error occurs during the Chain, it is returned as an exit status to the parent of the process performing the Chain.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`priority` is the initial priority of the process.

`path_cnt` specifies the number of I/O paths to copy (inherit).

`mod_name` points to the new program to execute.

`params` points to the parameter string to pass to the new process.

`mem_size` specifies the additional memory size above the default specified in the primary module's module header.

`param_size` specifies the size of the parameter string.

`type_lang` specifies the desired module type/language. `type_lang` must be either program/object or zero (for any).

Possible Errors

`EOS_NEMOD`

See Also

`F_CHAINM`
`F_FORK`
`F_FORKM`
`F_LOAD`

F_CHAINM

Execute New Primary Module Given Pointer to Module

Headers

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

Parameter Block Structure

```
typedef struct f_chainm_pb {
   syscb cb;
   u_int16 priority,
      path_cnt;
   Mh_com mod_head;
   u_char *params;
   u_int32 mem_size,
      param_size;
} f_chainm_pb, *F_chainm_pb;
```

Description

F_CHAINM executes a new program without the overhead of creating a new process. It is functionally similar to a F_FORK command followed by an F_EXIT. F_CHAINM resets the calling process' program and data memory areas and begins executing a new primary module. Open paths are not closed or otherwise affected.

F_CHAINM is similar to F_CHAIN. However, F_CHAINM is passed a pointer to the module instead of the module name.

F_CHAINM executes as follows:

1. The process' old primary module is unlinked.
2. The system tries to link to the module pointed to by the module header pointer.
3. The data memory area is reconfigured to the specified size in the new primary module's header.
4. Intercepts and pending signals are erased.

5. The following structure definition is passed to the initial function of the new module (this is identical to `F_FORK`).

```
typedef struct {
  process_id  proc_id;    /* process ID */
  owner_id    owner;      /* group/user ID */
  prior_level priority;   /* priority */
  u_int16     path_count; /* number of I/O paths
                                  inherited */
  u_int32     param_size, /* size of parameters */
              mem_size;   /* total initial memory
                                  allocated */
  u_char      *params,    /* parameter pointer */
              *mem_end;   /* top of memory pointer */
  Mh_com       mod_head; /*primary (forked) module ptr*/
} fork_params, *Fork_params;
```

The minimum overall data area size is 256 bytes.

An error is returned only if there is not enough memory to hold the parameters. If an error occurs during the `Chainm`, it is returned as an exit status to the parent of the process performing the `Chainm`.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`priority` is the initial priority of the process.

`path_cnt` is the number of I/O paths to copy (inherit).

`mod_head` points to the module header.

`params` points to the parameter string to pass to the new process.

`mem_size` specifies the additional memory size above the default specified in the primary module's module header.

`param_size` specifies the size of the parameter string.

Possible Errors

EOS_CRC

See Also

F_CHAIN
F_FORK
F_FORKM
F_LOAD

F_CHKMEM

Check Memory Block’s Accessibility

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_chkmem_pb {
   syscb cb;
   u_int32 size;
   u_int16 mode;
   u_char *mem_ptr;
   Pr_desc proc_desc;
} f_chkmem_pb, *F_chkmem_pb;
```

Description:

F_CHKMEM is called by system routines before 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 can access memory (such as I_READ and I_WRITE) to determine if the user process 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 the following:

- The calling process cannot use this service to check for write-only memory because it assumes read-only as the minimum. To check for no-access to a segment of memory, the calling process can check for read access and ensure the resulting status code is EOS_BPADDR. To check for read-only access, there must be two calls to F_CHKMEM.
- F_CHKMEM is only useful on systems with an MMU and having the SSM module in their bootfile. When SSM is active, the operating system validates all arguments. On systems without SSM, the call always returns successful because every process has full access rights to the entire memory space.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb`	is the control block header.
`size`	specifies the size of the memory area.
`mode`	specifies the permissions to check.
`mem_ptr`	points to the beginning of the memory to check.
`proc_desc`	points to the process descriptor of the target process.

Possible Errors

EOS_BPADDR
EOS_UNKSVC (from user state, with or without SSM)

See Also

`F_ALLTSK`
`F_DELTSK`
`F_PERMIT`
`F_PROTECT`
`I_READ`
`I_WRITE`

F_CHMDIR

Change Process' Current Module Directory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_chmdir_pb {
   syscb cb;
   u_char *name;
} f_chmdir_pb, *F_chmdir_pb;
```

Description

F_CHMDIR changes a process' current module directory.

The calling process must have access permission to the specified module directory or an EOS_PERMIT error is returned.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb

is the control block header.

name

points to the new current module directory. name can be a full pathlist or relative to the current module directory. To change to the system's root module directory, specify a slash (/) for name.

Possible Errors

EOS_BNAM
EOS_MNF
EOS_PERMIT

See Also

F_MKMDIR

F_CLRSIGS

Clear Process Signal Queue

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_clrsigs_pb {
   syscb cb;
   process_id proc_id;
} f_clrsigs_pb, *F_clrsigs_pb;
```

Description

F_CLRSIGS removes any pending signals sent to the target process.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	identifies the target process.

Possible Errors

EOS_IPRCID

See Also

F_SIGMASK

F_CMDPERM

Change Permissions of Module Directory

Headers

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

Parameter Block Structure

```
typedef struct f_cmdperm_pb {
   syscb cb;
   u_char *name;
   u_int16 perm;
} f_cmdperm_pb, *F_cmdperm_pb;
```

Description

F_CMDPERM changes the access permissions of an existing module directory. This makes it possible to restrict access to a particular module directory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
name	points to the name of the existing module directory.
perm	specifies the new permissions.

Possible Errors

EOS_BNAM
EOS_MNF
EOS_PERMIT

See Also

F_MKMDIR

F_CMPNAM

Compare Two Names

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_cmpnam_pb {
   syscb cb;
   u_int32 length;
   u_char *string,
      *pattern;
   int32 result;
} f_cmpnam_pb, *F_cmpnam_pb;
```

Description

F_CMPNAM compares a target name to a source pattern to determine if they are equal. F_CMPNAM is not case-sensitive; it does not differentiate between upper and lower case characters.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

length
specifies the length of the pattern string.

string
points to the target name string. The target name must be terminated by a null byte.

pattern
points to the pattern string. Two wildcard characters are recognized in the pattern string:

- A question mark (?) matches any single character.
- An asterisk (*) matches any string.

`result` is a returned value. It is the lexicographic result of the comparison.

- If `result` is zero, the target string is the same as the pattern string.
- If `result` is negative, the target name is greater than the pattern string.
- If `result` is positive, the target string is less than the pattern string.

Possible Errors

```
EOS_DIFFER
EOS_STKOVF
```

F_CONFIG

Configure an Element

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_config_pb {
   syscb cb;
   u_int32 code;
   void *param;
} f_config_pb, *F_config_pb;
```

Description

F_CONFIG is a wildcard call that configures elements of the operating system that may or may not be process specific. It dynamically reconfigures operating system resources at runtime. The target resources may be system-wide resources or they may be process-specific, depending on the nature of the configuration call being made.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

code
identifies the target configuration code. Currently, no sub-codes are defined for this call.

*param
points to any additional parameters required by the specified configuration function.

See Also

I_CONFIG

F_CPYMEM

Copy External Memory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_cpymem_pb {
   syscb cb;
   process_id proc_id;
   u_char *from,
      *to;
   u_int32 count;
} f_cpymem_pb, *F_cpymem_pb;
```

Description

F_CPYMEM uses F_MOVE to copy data from one location to another and (at present) ignores the proc_id argument (refer to the Parameters section below). The difference between F_MOVE and F_CPYMEM is the OS allows only system-state processes to use the former, while the OS allows either user- or system-state processes to use the later.

For system-state processes, the only difference between these two services is F_CPYMEM is slightly slower, since it has more routines to call before transferring control to F_MOVE.

For user-state processes, F_CPYMEM is the only choice for copying restricted memory.

The OS (if the SSM is active) calls F_CHKMEM to ensure the caller has read and write access to the output. The OS allows the input address to be any existent location of memory (it allows user-state processes to copy even restricted data if it exists in RAM).

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` specifies the process ID of the owner of the external memory.

This service does not currently use the `proc_id` input, which was valid when OS-9 was running on the MC6809 architecture. To allow memory access beyond 64KB, OS-9 used `F_CPYMEM` to do **bank switching** in order to allow a process to copy data from a different bank of memory. The `proc_id` argument was nothing more than a bank selector. At this point there is no need for the `proc_id` argument, but it is reserved for future use.

`from` is the address of the external process' memory to copy.

`to` points to the caller's destination buffer.

`count` is the number of bytes to copy.

Possible Errors

`EOS_BPADDR`

See Also

`F_MOVE`

F_CRC

Generate CRC

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_crc_pb {
   syscb cb;
   u_char *start;
   u_int32 count,
      accum;
} f_crc_pb, *F_crc_pb;
```

Description

F_CRC generates or checks the CRC (cyclic redundancy check) values of sections of memory. Compilers, assemblers, and 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 module (excluding the CRC). The accumulated CRC is complemented and stored in the correct position in the module.

The CRC is calculated over a specified number of bytes starting at the source address. It is not necessary to cover an entire module in one call, because the CRC may be accumulated over several calls. The CRC accumulator must be initialized to 0xffffffff before the first F_CRC call for any particular module.

To generate the CRC of an existing module, the calculation should be performed 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 module.h as CRCCON. The value is 0x00800fe3.

To generate the CRC for a module complete the following steps:

Step 1. Initialize the accumulator to `-1`.

Step 2. Perform the CRC over the module.

Step 3. Call `F_CRC` with a `NULL` value for `start`.

Step 4. Complement the CRC accumulator.

Step 5. Write the contents of the accumulator to the module.

The CRC value is three bytes long, in a four-byte field. To generate a valid module CRC, you must include the byte preceding the CRC in the check. You must initialize this byte to zero. For convenience, if a data pointer of zero is passed, the CRC is updated with one zero data byte. `F_CRC` always returns 0xff in the most significant byte of the `accum` parameter, so `accum` may be directly stored (after complement) in the last four bytes of a module as the correct CRC.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`start` points to the data.

`count` specifies the byte count for the data.

`accum` is a returned value. It points to the CRC accumulator.

See Also

`F_SETCRC`

F_CRLK

Create New Resource Lock Descriptor

Headers

```
#include <lock.h>
```

Parameter Block Structure

```
typedef struct f_crlk_pb {
   syscb cb;
   lock_id lid;
} f_crlk_pb, *F_crlk_pb;
```

Description

F_CRLK creates a new resource lock descriptor. A resource lock descriptor is allocated and initialized to a free state (not currently owned).

Locks can be used to protect resources in a multi-tasking environment. They provide a mechanism for exclusive access to a given resource.

Refer to **Chapter 6: Resource Locking** for more information on locks.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`lid` is a returned value. It is the lock identifier for the lock descriptor. `lid` is used as a handle to perform further operations on the lock.

Possible Errors

EOS_NORAM

See Also

F_ACQLK
F_CAQLK
F_DELLK
F_RELLK
F_WAITLK

F_DATMOD

Create Data Module

Headers

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

Parameter Block Structure

```
typedef struct f_datmod_pb {
   syscb cb;
   u_char *mod_name;
   u_int32 size;
   u_int16 attr_rev,
      type_lang,
      perm;
   void *mod_exec;
   Mh_com mod_head;
   u_int32color
} f_datmod_pb, *F_datmod_pb;
```

Description

F_DATMOD creates a data module with the specified attribute/revision and clears the data portion of the module. The module is created and entered into the current module directory. Several processes can communicate with each other using a shared data module.

Be careful not to alter the data module’s header or name string to prevent the module from becoming unknown to the system.

The created module contains at least size 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 is rounded up to the nearest system memory allocation boundary.

F_DATMOD does not create a CRC value for the data module. If you load the data module into memory, you must first create the CRC value.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`mod_name` points to the module name string.

`size` is the size of the data portion.

`attr_rev` is a returned value. It is the value of the module's attribute and revision.

`type_lang` is a returned value. It is the value of the module's type and language.

`perm` specifies the access permissions for the module.

`mod_exec` is a returned value. It points to the module data.

`mod_head` is a returned value. It points to the module header.

`color` is the memory color type. If color is zero, `MEM_ANY` is the specified memory type.

Possible Errors

`EOS_BNAM`
`EOS_KWNMOD`

See Also

`F_SETCRC`

F_DATTACH

Attach Debugger to a Running Process

Headers

```
#include <regs.h>
```

Parameter Block Structure

```
typedef struct f_dattach_pb {
   syscb cb;
   process_id proc_id;
   Regs reg_stack;
   Fregs freg_stack;
} f_dattach_pb, *F_dattach_pb;
```

Description

F_DATTACH attaches the calling debugger to an active process, enabling the debugger to assume debug control over the existing process. It establishes a debug session in the same way F_DFORK starts a new process for debug execution. Once a debugger performs the debug attach operation, the target process is suspended from execution and the debugger can then proceed to execute the target process under its control using the F_DEXEC service request. One important difference between F_DATTACH and F_DFORK is with the F_DATTACH call, the target process continues normal execution when the parent debugging process exits. The debug resources of the target process are released but the process does not terminate. However, when a process is started with the F_DFORK service request, the process is terminated when the parent debugger performs the F_DEXIT service request.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` is the process identifier of the target process to attach to for debugging.

`reg_stack` points to a register image buffer in the caller's data area where the kernel returns the current register image of the target debug process.

`freg_stack` points to a floating-point register image buffer in the caller's data area where the kernel returns the current floating-point register image of the target debug process. Note, this floating-point image can contain an empty image since the target process may not be using the floating-point facilities of the system.

Possible Errors

EOS_IPRCID
EOS_PERMIT

See Also

F_DEXEC
F_DEXIT
F_DFORK

F_DDLK

Check for Deadlock Situation

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_ddlk_pb {
   syscb cb;
   process_id proc_id;
} f_ddlk_pb, *F_ddlk_pb;
```

Description

`F_DDLK` checks for a deadlock situation between processes. A search for the current process (calling process) in the linked list of potential conflicting processes is begun from the process specified by `proc_id`.

`F_DDLK` is useful for preventing a deadlock situation when protecting multiple resources from simultaneous accesses. The deadlock list usually represents the list of processes waiting to acquire access to an associated resource.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` specifies the process with which to begin the search.

If the calling process is already in the linked list of processes, an `EOS_DEADLK` error is returned to the caller.

If the process is not in the linked list, the current process is added to the list associated with `proc_id`.

Possible Errors

`EOS_DEADLK`

F_DELLK

Delete Existing Lock Descriptor

Headers

```
#include <lock.h>
```

Parameter Block Structure

```
typedef struct f_dellk_pb {
    syscb cb;
    lock_id lid;
} f_dellk_pb, *F_dellk_pb;
```

Description

F_DELLK deletes an existing lock descriptor.

F_DELLK does not check for suspended processes still waiting to acquire the lock; an implementation using locks must release all processes waiting on a resource lock prior to deleting it. You can corrupt the system if you do not release suspended processes prior to deletion.

Refer to Chapter 6 or more information about locks.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb	is the control block header.
lid	is the lock identifier for the lock to delete.

See Also

F_ACQLK
F_CAQLK
F_CRLK
F_RELLK
F_WAITLK

F_DELMDIR

Delete Existing Module Directory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_delmdir_pb {
   syscb cb;
   u_char *name;
} f_delmdir_pb, *F_delmdir_pb;
```

Description

F_DELMDIR deletes an existing module directory.

If the target module directory is not empty, an EOS_DNE **directory not empty** error is returned.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
name	points to the module directory.

Possible Errors

```
EOS_BNAM
EOS_DNE
EOS_MNF
EOS_PERMIT
```

F_DELTSK

Deallocate Process Descriptor

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_deltsk_pb {
   syscb cb;
   Pr_desc proc_desc;
} f_deltsk_pb, *F_deltsk_pb;
```

Description

`F_DELTSK` is called when a process terminates to return the process' protection resources. This call must release any protection structures allocated to the process, whether this be memory or any hardware resource.

`F_DELTSK` is only supported on systems with a memory protection unit (for example, all 80386 and 80486 systems and PowerPC systems). The SSM module **must** be present in the bootfile.

If the SSM module is not present in the system, an `EOS_UNKSVC` error is returned. You should ignore this error.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb`	is the control block header.
`proc_desc`	points to the process descriptor.

Possible Errors

```
EOS_BNAM
EOS_UNKSVC
```

See Also

F_ALLTSK
F_CHKMEM
F_PERMIT
F_PROTECT

F_DEXEC

Execute Debugged Program

Headers

```
#include <types.h>
#include <dexec.h>
```

Parameter Block Structure

```
typedef struct f_dexec_pb {
   syscb cb;
   process_id proc_id;
   dexec_mode mode;
   u_char *params;
   u_int32 num_instr,
      tot_instr,
      except,
      addr;
   u_int16 num_bpts,
      **brk_pts;
   dexec_status status;
   error_code exit_status;
} f_dexec_pb, *F_dexec_pb;
```

Description

F_DEXEC controls the execution of a suspended child process created by F_DFORK. The process performing the F_DEXEC is suspended, and its debugged child process is executed instead. This process terminates and control returns to the parent after the specified number of instructions have been executed, a breakpoint is reached, or an unexpected exception occurs. Therefore, the parent and the child processes are never active at the same time.

When F_DEXEC is executed in DBG_M_SOFT or DBG_M_COUNT mode, it traces every instruction of the child process and checks for the termination conditions. Breakpoints are lists of addresses to check and work with ROMed object programs. Consequently, the child process being debugged runs at a slow speed.

When F_DEXEC is executed in DBG_M_HARD mode, it replaces the instruction at each breakpoint address with an illegal opcode. Next, it 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. However, F_DEXEC cannot count the number of executed instructions.

When status is DBG_S_EXCEPT, the except parameter is the specific exception identifier (error) causing the child to return to the debugger.

OS-9 system calls made by the suspended program are executed at full speed and are considered one logical instruction. This is also true of system-state trap handlers. You cannot debug system-state processes.

The system uses the register buffer and floating point 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.

An F_DEXIT call must be made to return the debugged process' resources (memory).

Tracing is allowed through subroutine libraries and intercept routines. This is not a problem, but you will see code executed that is not explicitly in your sources.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	is the process ID of the child to execute.

mode
specifies the debug mode. These modes are defined in the header file dexec.h:

Debug Modes	Description
DBG_M_INACTV	Inactive mode (used by the kernel).
DBG_M_HARD	Hard breakpoints/full speed execution.
DBG_M_SOFT	Soft breakpoints/continuous execution.
DBG_M_COUNT	Execute **count** instructions.
DBG_M_CONTROL	Execute until change of control (future release).

params
is the parameter list pointer. This will be implemented in a future release.

num_instr
is the number of instructions to execute. If num_instr is zero, commands are executed continuously. Upon completion of the F_DEXEC call, num_instr is updated with a value representing the original number of instructions less the number of instructions executed.

tot_instr
is a returned value. It points to the number of instructions executed so far when the child is executed in trace mode.

except
is a returned value. It is the exception the child received, when the child process returned due to an exception.

addr
is a returned value. It is the violation address associated with an exception condition.

num_bpts
specifies the number of breakpoints in the list.

brk_pts

points to the breakpoint list. The breakpoint list is a list of addresses indicating which instructions are considered breakpoints.

status

is the process return status. status indicates the reason the child process returned to the debugger. The following status modes are defined in the header file dexec.h:

Status Modes	Description
DBG_S_FINISH	The command finished successfully.
DBG_S_BRKPNT	The process hit a breakpoint.
DBG_S_EXCEPT	An exception occurred during execution.
DBG_S_CHILDSIG	The process received a signal (no intercept).
DBG_S_PARENTSIG	The debugger received a signal.
DBG_S_CHAIN	The process made an F_CHAIN system call.
DBG_S_EXIT	The process made an F_EXIT system call.
DBG_S_CONTROL	The process executed a jmp or bra (future release).
DBG_S_WATCH	The process hit a watch point (future release).
DBG_S_FORK	The process made an F_FORK system call.

exit_status

is a returned value. It is the child's exit status, when the child performs an F_EXIT call.

Possible Errors

EOS_IPRCID
EOS_PRCABT

See Also

F_CHAIN
F_DEXIT
F_DFORK
F_EXIT

F_DEXIT

Exit Debugged Program

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_dexit_pb {
   syscb cb;
   process_id proc_id;
} f_dexit_pb, *F_dexit_pb;
```

Description

F_DEXIT terminates a suspended child process created by F_DFORK. The F_EXIT done by the child process does not release the child's resources in the case of a debugged process. This enables examination of the child after its termination. Therefore, the debugger must do an F_DEXIT to release the child's resources after this call.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	is the process ID of the child to terminate.

Possible Errors

EOS_IPRCID

See Also

F_DEXEC
F_DFORK
F_EXIT

F_DFORK

Fork Process Under Control of Debugger

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_dfork_pb {
   syscb cb;
   u_int16 priority,
      path_cnt;
   process_id proc_id;
   Regs reg_stack;
   Fregs freg_stack;
   u_char *mod_name,
      *params;
   u_int32 mem_size,
      param_size;
   u_int16 type_lang;
} f_dfork_pb, *F_dfork_pb;
```

Description:

F_DFORK creates a new process that becomes a child of the caller. It sets up the process' memory, MPU registers, and standard I/O paths. In addition, F_DFORK enables a debugger utility to create a process whose execution can be closely controlled. The created process is not placed in the active queue, but is left in a suspended state. This enables the debugger to control its execution through the F_DEXEC and F_DEXIT system calls.

The child process is created in the DBG_M_SOFT (trace) mode and is executed with F_DEXEC.

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 F_DEXEC.

For information about process creation, refer to the F_FORK description.

Processes whose primary module is owned by a super-user can only be debugged by a super user. You cannot debug system-state processes.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb is the control block header.

priority is the priority of the new process.

path_cnt is the number of I/O paths for the child to inherit.

proc_id is a returned value. It is the new child process ID.

reg_stack points to the register buffer.

freg_stack points to the floating point register buffer.

mod_name points to the module name.

params points to the parameter string to pass to the new process.

mem_size specifies any additional stack space to allocate above the default specified in the primary module’s module header.

param_size specifies the size of the parameter string.

type_lang specifies the desired type and language of the primary module to be executed.

Possible Errors

EOS_MNF
EOS_NEMOD
EOS_NORAM
EOS_PERMIT
EOS_PNNF

See Also

F_DEXEC
F_DEXIT
F_DFORKM
F_FORK

F_DFORKM

Fork Process Under Control of Debugger

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_dforkm_pb {
   syscb cb;
   u_int16 priority,
      path_cnt;
   process_id proc_id;
   Regs reg_stack;
   Fregs freg_stack;
   Mh_com mod_head;
   u_char *params;
   u_int32 mem_size,
      param_size;
} f_dforkm_pb, *F_dforkm_pb;
```

Description

F_DFORKM creates a new process that becomes a child of the caller. It sets up the process' memory, MPU registers, and standard I/O paths. In addition, F_DFORKM enables a debugger utility to create a process whose execution can be closely controlled. The created process is not placed in the active queue, but is left in a suspended state. This enables the debugger to control its execution through the F_DEXEC and F_DEXIT system calls. F_DFORKM is similar to F_DFORK. However, F_DFORKM is passed a pointer to the module to fork rather than the module name.

For more information, refer to the description of F_DFORK.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`priority` is the priority of the new process.

`path_cnt` is the number of I/O paths for the child to inherit.

`proc_id` is a returned value. It is a the new child process ID.

`reg_stack` points to the register buffer.

`freg_stack` points to the floating point register buffer.

`mod_head` points to the module header.

`params` points to the parameter string to pass to the new process.

`mem_size` specifies any additional stack space to allocate above the default specified in the primary module's module header.

`param_size` specifies the size of the parameter string.

Possible Errors

EOS_MNF
EOS_NEMOD
EOS_NORAM
EOS_PERMIT
EOS_PNNF

See Also

F_DEXEC
F_DEXIT
F_DFORK
F_FORK

F_EVENT

Process Synchronization and Communication

Headers

Refer to the specific event for the header to include.

Parameter Block Structure

Refer to the specific event for the appropriate parameter block structure.

Description

OS-9 events are multiple-value semaphores that synchronize concurrent processes sharing resources such as files, data modules, and CPU time. The events' functions enable processes to create/delete events, link/unlink events, get event information, and suspend operation until an event occurs. Events are also used for various means of signalling.

The following events functions are currently supported:

Event	Description
`F_EVENT, EV_ALLCLR`	Wait for all bits defined by mask to become clear.
`F_EVENT, EV_ALLSET`	Wait for all bits defined by mask to become set.
`F_EVENT, EV_ANYCLR`	Wait for any bits defined by mask to become clear.
`F_EVENT, EV_ANYSET`	Wait for any bits defined by mask to become set.
`F_EVENT, EV_CHANGE`	Wait for any bits defined by mask to change.
`F_EVENT, EV_CREAT`	Create new event.
`F_EVENT, EV_DELET`	Delete existing event.
`F_EVENT, EV_INFO`	Return event information.
`F_EVENT, EV_LINK`	Link to existing event by name.
`F_EVENT, EV_PULSE`	Signal event occurrence.
`F_EVENT, EV_READ`	Read event value without waiting.
`F_EVENT, EV_SET`	Set event variable and signal event occurrence.

Event	Description
`F_EVENT, EV_SETAND`	Set event value by ANDing the event value with a mask.
`F_EVENT, EV_SETOR`	Set event value by ORing the event value with a mask.
`F_EVENT, EV_SETR`	Set relative event variable and signal event occurrence.
`F_EVENT, EV_SETXOR`	Set event value by XORing the event value with a mask.
`F_EVENT, EV_SIGNL`	Signal event occurrence.
`F_EVENT, EV_TSTSET`	Wait for all bits defined by mask to clear, then set these bits.
`F_EVENT, EV_UNLNK`	Unlink event.
`F_EVENT, EV_WAIT`	Wait for event to occur.
`F_EVENT, EV_WAITR`	Wait for relative event to occur.

Specific parameters and functions of each event operation are discussed in the following pages. The `EV_XXX` function names are defined in the system definition file `funcs.h`.

For more information on events, refer to Chapter 3.

The event value is added to `min_val` and `max_val`, and the actual values are returned to the caller. If an underflow or overflow occurs on the addition, the values 0x80000000 (minimum integer) and 0x7fffffff (maximum integer) are used, respectively.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Possible Errors

`EOS_EVNTID`

See Also

`F_EVENT, EV_SIGNL`

F_EVENT, EV_ALLCLR

Wait for All Bits Defined by Mask to Become Clear

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evallclr_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 mask;
} f_evallclr_pb, *F_evallclr_pb;
```

Description

EV_ALLCLR waits until one of the event *set* calls occurs that clears all of the bits corresponding to the set bits in the mask. The event variable is ANDed with the value in mask. If the resulting value is not zero, the calling process is suspended in a FIFO event queue.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_ALLCLR event function code.
ev_id	identifies the event.

`value` is a returned value. It is the actual event value after the *set* operation that activated the suspended process.

If the process receives a signal while in the event queue, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned and the caller's intercept routine is executed.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

Possible Errors

```
EOS_EVNTID
EOS_SIGNAL
```

F_EVENT, EV_ALLSET

Wait for Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evallset_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 mask;
} f_evallset_pb, *F_evallset_pb;
```

Description

`EV_ALLSET` waits until one of the event *set* calls occurs that sets all of the bits corresponding to the set bits in the mask. The event variable is ANDed with the value in `mask` and then EXCLUSIVE-ORed with it. If the resulting value is not zero, the calling process is suspended in a FIFO event queue.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_ALLSET` event function code.
`ev_id`	identifies the event.

`value` is a returned value. It is the actual event value after the *set* operation that activated the suspended process.

If the process receives a signal while in the event queue, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned and the caller's intercept routine is executed.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

Possible Errors

`EOS_EVNTID`
`EOS_SIGNAL`

F_EVENT, EV_ANYCLR

Wait for Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evanyclr_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 mask;
} f_evanyclr_pb, *F_evanyclr_pb;
```

Description

`EV_ANYCLR` waits for an event to occur. The event variable is ANDed with the value in `mask` and then EXCLUSIVE-ORed with it. If the resulting value is zero, the calling process is suspended in a FIFO event queue. It waits until one of the event *set* calls occurs that clears any of the bits corresponding to the set bits in the mask.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_ANYCLR` event function code.
`ev_id`	identifies the event.

`value` is a returned value. It is the actual event value after the *set* operation that activated the suspended process.

If the process receives a signal while in the event queue, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned and the caller's intercept routine is executed.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

Possible Errors

```
EOS_EVNTID
EOS_SIGNAL
```

F_EVENT, EV_ANYSET

Wait for Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evanyset_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 mask;
} f_evanyset_pb, *F_evanyset_pb;
```

Description

`EV_ANYSET` waits until one of the event *set* calls occurs that sets any of the bits corresponding to the set bits in the mask. The event variable is ANDed with the value in `mask`. If the resulting value is zero, the calling process is suspended in a FIFO event queue.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_ANYSET` event function code.
`ev_id`	identifies the event.

`value` is a returned value. It is the actual event value after the *set* operation that activated the suspended process.

If the process receives a signal while in the event queue, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned and the caller’s intercept routine is executed. The signal code is returned in `signal`.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

Possible Errors

```
EOS_EVNTID
EOS_SIGNAL
```

F_EVENT, EV_CHANGE

Wait for Event to Occur

Headers:

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evchange_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 mask;
   u_int32 pattern;
} f_evchange_pb, *F_evchange_pb;
```

Description

`EV_CHANGE` waits until one of the event *set* calls occurs that changes any of the bits corresponding to the set bits in `mask`. The event variable is ANDed with the value in `mask`. If the resulting value is not equal to the wait pattern, the calling process is suspended in a FIFO event queue.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_CHANGE` event function code.
`ev_id`	identifies the event.

`value` is a returned value. It is the actual event value after the *set* operation that activated the suspended process.

If the process receives a signal while in the event queue, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned and the caller's intercept routine is executed. The signal code is returned in `signal`.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

`pattern` specifies the wait pattern.

Possible Errors

```
EOS_EVNTID
EOS_SIGNAL
```

F_EVENT, EV_CREAT

Create New Event

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evcreat_pb {
   syscb cb;
   u_int16 ev_code,
      wait_inc,
      sig_inc,
      perm,
      color;
   event_id ev_id;
   u_char *ev_name;
   u_int32 value;
} f_evcreat_pb, *F_evcreat_pb;
```

Description

EV_CREAT creates events dynamically as needed. When an event is created, an initial value is specified, as well as increments to be applied each time the event is waited for or occurs. Subsequent event calls use the returned ID number to refer to the created event.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

ev_code
is the EV_CREAT event function code.

wait_inc
specifies the auto-increment value for EV_WAIT.

sig_inc
specifies the auto-increment value for EV_SIGNL.

perm
specifies the access permissions.

`color` specifies the memory type for the event structure.

`ev_id` is a returned value. It is the event identifier used for subsequent event calls.

`ev_name` points to the event name string.

`value` specifies the initial event variable value.

Possible Errors

EOS_BNAM
EOS_EVBUSY
EOS_EVFULL
EOS_NORAM

See Also

F_EVENT, EV_DELET
F_EVENT, EV_SIGNL
F_EVENT, EV_WAIT

F_EVENT, EV_DELET

Remove Event

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evdelet_pb {
   syscb cb;
   u_int16 ev_code;
   u_char *ev_name;
} f_evdelet_pb, *F_evdelet_pb;
```

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), which is incremented with each EV_LINK call and decremented with each EV_UNLINK call. An event may not be deleted unless its use count is zero.

OS-9 does not automatically *unlink* events when EOS_EXIT occurs.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_DELET event function code.
name	points to the event's name string.

Possible Errors

EOS_BNAM
EOS_EVBUSY
EOS_EVNF

See Also

`F_EVENT, EV_CREAT`
`F_EVENT, EV_LINK`
`F_EVENT, EV_UNLNK`

F_EVENT, EV_INFO

Return Event Information

Headers

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

Parameter Block Structure

```
typedef struct f_evinfo_pb {
    syscb cb;
    u_int16 ev_code;
    event_id ev_id;
    u_int32 size;
    u_char *buffer;
} f_evinfo_pb, *F_evinfo_pb;
```

Description

`EV_INFO` returns event information in your buffer. This call is used by utilities needing to know the status of all active events. The information returned is defined by the `ev_infostr` event information structure defined in the `events.h` header file.

Refer to Chapter 3 for more information on events.

The name of the event is appended to the end of the information structure. The information `buffer` and `size` parameters must be large enough to accommodate the name of the target event.

`EV_INFO` returns the event information block for the first active event whose index is greater than or equal to this index. If no such event exists, an error is returned.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_INFO` event function code.
`ev_id`	specifies the event index to use to begin the search. Unlike other event functions, only an event index is passed in the `ev_id` parameter. The index is the system event number, ranging from zero to one less than the maximum number of system events.
`size`	specifies the buffer size.
`buffer`	points to the buffer containing the event information.

Possible Errors

`EOS_EVNTID`

See Also

`ev_str/ev_infostr`

F_EVENT, EV_LINK

Link to Existing Event by Name

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evlink_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   u_char *ev_name;
} f_evlink_pb, *F_evlink_pb;
```

Description

EV_LINK determines the ID number of an existing event. Once an event has been linked, all subsequent references are made using the returned event ID. This permits the system to access events quickly, while preventing programs from using invalid or deleted events. The event use count is incremented when an EV_LINK is performed. To keep the use count synchronized properly, use EV_UNLINK when the event is no longer used.

The event access permissions are checked only at link time.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_LINK event function code.
ev_name	points to the event name string.
ev_id	is the event identifier used for subsequent event calls.

Possible Errors

EOS_BNAM
EOS_EVNF
EOS_PERMIT

See Also

F_EVENT, EV_UNLNK

F_EVENT, EV_PULSE

Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evpulse_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   u_int32 actv_flag;
} f_evpulse_pb, *F_evpulse_pb;
```

Description

`EV_PULSE` signals an event occurrence. The event value is set to what is passed in `value`, and the signal auto-increment is not applied. Then, the event queue is searched for the first process waiting for that event value, after which the original event value is restored.

`EV_PULSE` with the `actv_flag` set executes as follows for each process in the queue until the queue is exhausted:

1. The signal auto-increment is added to the event variable.
2. The first process in range is awakened.
3. The event value is updated with the wait auto-increment.
4. The search is continued with the updated value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_PULSE` event function code.

`ev_id` identifies the event.

`value` is the event value prior to the pulse operation.

`actv_flag` specifies which process(es) to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_READ

Read Event Value Without Waiting

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evread_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
} f_evread_pb, *F_evread_pb;
```

Description

EV_READ reads the value of an event without waiting or affecting the event variable. This can determine the availability of the event (or associated resource) without waiting.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_READ event function code.
ev_id	identifies the event.
value	is the current event value.

Possible Errors

EOS_EVNTID

F_EVENT, EV_SET

Set Event Variable and Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evset_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   u_int32 actv_flag;
} f_evset_pb, *F_evset_pb;
```

Description

EV_SET signals an event has occurred. The current event variable is set to the value passed at value, rather than updated with the signal auto-increment. Next, the event queue is searched for the first process waiting for the event value.

EV_SET with the actv_flag set executes as follows for each process in the queue until the queue is exhausted:

1. The first process in range is awakened.
2. The event value is updated with the wait auto-increment.
3. The search is continued with the updated value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_SET event function code.
ev_id	identifies the event.

`value`	is the event value prior to the set operation.
`actv_flag`	specifies which process(es) to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_SETAND

Set Event Variable and Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evsetand_pb {
  syscb cb;
  u_int16 ev_code;
  event_id ev_id;
  u_int32 mask,
    actv_flag;
} f_evsetand_pb, *F_evsetand_pb;
```

Description

`EV_SETAND` signals an event has occurred. The current event variable is ANDed with the value passed in `mask` rather than updated with the signal auto-increment. Next, the event queue is searched for the first process waiting for that event value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`
is the control block header.

`ev_code`
is the `EV_SETAND` event function code.

`ev_id`
identifies the event.

`value`
is the event value prior to the logical operation.

`mask`	is the event mask. It indicates which bits are significant to the caller.
`actv_flag`	specifies which process(es) to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_SETOR

Set Event Variable and Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evsetor_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   u_int32 mask,
      actv_flag;
} f_evsetor_pb, *F_evsetor_pb;
```

Description

`EV_SETOR` signals an event has occurred. The current event variable is ORed with the value passed in `mask`. Next, the event queue is searched for the first process waiting for that event value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`ev_code` is the `EV_SETOR` event function code.

`ev_id` identifies the event.

`value` is the event value prior to the logical operation.

`mask` is the event mask. It indicates which bits are significant to the caller.

`actv_flag` specifies which processes to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_SETR

Set Relative Event Variable and Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evsetr_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   u_int32 actv_flag;
} f_evsetr_pb, *F_evsetr_pb;
```

Description

`EV_SETR` signals an event has occurred. The current event value is incremented by `value`, rather than by the signal auto-increment. Next, the event queue is searched for the first process waiting for that event value. Arithmetic underflows or overflows are set to 0x80000000 (minimum integer) or 0x7fffffff (maximum integer), respectively.

`EV_SETR` with the `actv_flag` set executes as follows for each process in the queue until the queue is exhausted:

1. The first process in range is awakened.
2. The event value is updated with the wait auto-increment.
3. The search is continued with the updated value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_SETR` event function code.

`ev_id` identifies the event.

`value` is the event value after the relative operation.

`actv_flag` specifies which process(es) to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

See Also

`F_EVENT, EV_SET`
`F_EVENT, EV_SIGNL`

F_EVENT, EV_SETXOR

Set Event Variable and Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evsetxor_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   u_int32 mask,
      actv_flag;
} f_evsetxor_pb, *F_evsetxor_pb;
```

Description

`EV_SETXOR` signals an event has occurred. The current event value is EXCLUSIVE-ORed with `mask` rather than updated with the signal auto-increment. Next, the event queue is searched for the first process waiting for that event value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`ev_code` is the `EV_SETXOR` event function code.

`ev_id` identifies the event.

`value` is the event value prior to the logical operation.

`mask` specifies the event mask. It indicates which bits are significant to the caller.

`actv_flag` specifies which process(es) to activate.

- If `actv_flag` is one, all processes in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_SIGNL

Signal Event Occurrence

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evsignl_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   u_int32 actv_flag;
} f_evsignl_pb, *F_evsignl_pb;
```

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. Next, the event queue is searched for the first process waiting for that event value.

`EV_SIGNL` with the `actv_flag` set, executes as follows for each process in the queue until the queue is exhausted:

1. The signal auto-increment is added to the event variable.
2. The first process in range is awakened.
3. The event value is updated with the wait auto-increment.
4. The search is continued with the updated value.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`ev_code`	is the `EV_SIGNL` event function code.

`ev_id` identifies the event that has occurred.

`value` is the event value prior to the signal operation.

`actv_flag` specifies which process(es) to activate.

- If `actv_flag` is one, all processes in the event queue with a value in range are activated.
- If `actv_flag` is not set, only the first process in the event queue waiting for that range is activated.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_TSTSET

Wait for Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evtstset_pb {
    syscb cb;
    u_int16 ev_code;
    event_id ev_id;
    int32 value;
    signal_code signal;
    u_int32 mask;
} f_evtstset_pb, *F_evtstset_pb;
```

Description

`EV_TSTSET` waits for an event to occur. The event variable is ANDed with the value in `mask`. If the result is not zero, the calling process is suspended in a FIFO event queue until an `EV_SIGNL` occurs clearing all of the bits corresponding to the set bits in the mask. Next, the bits corresponding to the set bits in the mask are set.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`ev_code` is the `EV_TSTSET` event function code.

`ev_id` identifies the event.

`value` is a returned value. It is the actual event value prior to the *set* operation that activates the suspended process.

If a process in the event queue receives a signal, it is activated and an `EOS_SIGNAL` error is returned, even though the event has not actually occurred. Also, the current event value is returned, and the caller's intercept routine is executed.

`signal` contains the returned signal code.

`mask` specifies the activation mask. It indicates which bits are significant to the caller.

Possible Errors

`EOS_EVNTID`

F_EVENT, EV_UNLNK

Unlink Event

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evunlnk_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
} f_evunlnk_pb, *F_evunlnk_pb;
```

Description

EV_UNLNK informs the system a process is no longer using the event. This decrements the event use count and allows the event to be deleted with the EV_DELET event function when the count reaches zero.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_UNLINK event function code.
ev_id	specifies the event.

Possible Errors

EOS_EVNTID

See Also

F_EVENT, EV_DELET
F_EVENT, EV_LINK

F_EVENT, EV_WAIT

Wait for Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evwait_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 min_val,
      max_val;
} f_evwait_pb, *F_evwait_pb;
```

Description

`EV_WAIT` waits until an event call places the value in the range between the minimum and maximum activation values. Next, the wait auto-increment (specified at creation) is added to the event variable.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`ev_code` is the `EV_WAIT` event function code.

`ev_id` identifies the event.

`value` is a returned value. It is the actual event value prior to the *set* operation that activates the suspended process.

`signal` is a returned value. It is the signal code, if it is activated by a signal. If a process in the event queue receives a signal, it is activated even though the event has not actually occurred. The auto-increment is not added to the event variable, and an `EOS_SIGNAL` error is returned. Also, the event value is returned, even though it is not in range, and the caller's intercept routine is executed.

`min_val` is the minimum activation value.

`max_val` is the maximum activation value. The event value is added to `min_val` and `max_val`, and the actual absolute values are returned to the caller. If an underflow or overflow occurs on the addition, the values 0x80000000 (minimum integer) and 0x7fffffff (maximum integer) are used, respectively.

Possible Errors

`EOS_EVNTID`

See Also

`F_EVENT, EV_SIGNL`
`F_EVENT, EV_WAIT`

F_EVENT, EV_WAITR

Wait for Relative Event to Occur

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_evwaitr_pb {
   syscb cb;
   u_int16 ev_code;
   event_id ev_id;
   int32 value;
   signal_code signal;
   u_int32 min_val,
      max_val;
} f_evwaitr_pb, *F_evwaitr_pb;
```

Description

EV_WAITR waits until an event call places the value in the range between the minimum and maximum activation values, where min_val and max_val are relative to the current event value. Next, the wait auto-increment (specified at creation) is added to the event variable.

The event value is added to min_val and max_val, and the actual absolute values are returned to the caller. If an underflow or overflow occurs on the addition, the values 0x80000000 (minimum integer) and 0x7fffffff (maximum integer) are used, respectively.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
ev_code	is the EV_WAITR event function code.
ev_id	identifies the event.

`value` is a returned value. It is the actual event value prior to the *set* operation that activates the suspended process.

`signal` is a returned value. It is the signal code, if it is activated by a signal.

If a process in the event queue receives a signal, it is activated even though the event has not actually occurred. The auto-increment is not added to the event variable, and an `EOS_SIGNAL` error is returned. Also, the event value is returned, even though it is not in range, and the caller's intercept routine is executed.

`min_val` is the minimum relative activation value. Upon return, it contains the absolute minimum activation value.

`max_val` is the maximum relative activation value. Upon return, it contains the absolute maximum activation value.

Possible Errors

`EOS_EVNTID`

See Also

`F_EVENT, EV_SIGNL`
`F_EVENT, EV_WAIT`

F_EXIT

Terminate Calling Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_exit_pb {
    syscb cb;
    status_code status;
} f_exit_pb, *F_exit_pb
```

Description

F_EXIT allows a process to terminate itself. Its data memory area is deallocated and its primary module is unlinked. All open paths are automatically closed.

The parent can detect the death of a child process by executing F_WAIT. This returns (to the parent) the exit status passed by the child in its exit call. The shell assumes the exit status is an OS-9 error code. The exit status can also be a user-defined status value.

Processes to be called directly by the shell should only return an OS-9 error code or zero (if no error occurred).

The parent **must** perform an F_WAIT or an F_EXIT before the child process descriptor is returned to free memory.

F_EXIT executes as follows:

1. Close all paths.
2. Return the memory to the system.
3. Unlink the primary module, subroutine libraries, and trap handlers.
4. Free the process descriptor of any dead child processes.
5. Free the process descriptor if the parent is dead.
6. Leave the process in limbo until the parent notices the death if the parent has not executed F_WAIT.
7. If the parent is waiting, move it to the active queue and informs it of death/status.

8. Remove the child from the sibling list and free its process descriptor memory.

Only the primary module, subroutine libraries, and trap handlers are unlinked. Any other modules loaded or linked by the process should be unlinked before calling `F_EXIT`.

Although `F_EXIT` closes any open paths, it ignores errors returned by `I_CLOSE`. Due to I/O buffering, write errors can go unnoticed when paths are left open. However, by convention, the standard I/O paths (0, 1, and 2) are usually left open.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`status`	is the status code returned to the parent process.

See Also

`F_APROC`
`F_FORK`
`F_SRTMEM`
`F_UNLINK`
`F_WAIT`
`I_CLOSE`

F_FINDPD

Find Process Descriptor

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_findpd_pb {
   syscb cb;
   process_id proc_id;
   Pr_desc proc_desc;
} f_findpd_pb, *F_findpd_pb;
```

Description

`F_FINDPD` converts a process number to the absolute address of its process descriptor data structure.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` specifies the process ID.

`proc_desc` is a returned value. It is the pointer to the process descriptor.

Possible Errors

`EOS_IPRCID`

See Also

`F_ALLPRC`
`F_RETPD`

F_FMOD

Find Module Directory Entry

Headers

```
#include <moddir.h>
```

Parameter Block Structure

```
typedef struct f_findmod_pb {
   syscb cb;
   u_int16 type_lang;
   Mod_dir moddir_entry;
   u_char *mod_name;
}  f_findmod_pb, *F_findmod_pb;
```

Description

F_FMOD searches the module directory for a module whose name, type, and language match the parameters. If found, a pointer to the module directory entry is returned in moddir_entry.

Attributes

Operating System:	OS-9
State:	System and Interrupt
Threads:	Safe

Parameters

cb
is the control block header.

type_lang
specifies the type and language of the module.

moddir_entry
is a returned value. It is the pointer to the module directory entry.

mod_name
points to the module name.

Possible Errors

EOS_BNAM
EOS_MNF

See Also

`F_LINK`
`F_LOAD`

F_FORK

Create New Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_fork_pb {
  syscb cb;
  u_int16 priority,
    path_cnt;
  process_id proc_id;
  u_char *mod_name,
    *params;
  u_int32 mem_size,
    param_size;
  u_int16 type_lang;
  u_int16 orphan;
} f_fork_pb, *F_fork_pb;
```

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 that is initially executed). If the program is found in the current or alternate module directory, the module is linked and executed. If the program is not found, 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. The module must be program object code with the appropriate read and/or execute permissions to be loaded successfully.

The primary module's header determines the process' initial data area size. OS-9 attempts to allocate RAM equal to the required data storage size, the size of any parameters passed, and the size specified by `mem_size`. The RAM area must be contiguous.

The execution offset in the module header is used to set the PC to the module's entry point.

When the shell processes a command line, it passes a copy of the command line parameters (if any) as a parameter string. The shell appends an end-of-line character to the parameter string to simplify string-oriented processing.

If one or more of these operations is unsuccessful, the fork is aborted and the caller receives an error.

F_FORK passes the following structure (defined in <fork.h>) as a parameter to the newly-created process:

```
typedef struct {
   process_id        proc_id;    /* process ID */
   owner_id          owner;      /* group/user ID */
   priority_level    priority;   /* priority */
   u_int16           path_count; /* number of I/O paths
inherited */
   u_int32           param_size, /* size of parameters */
                     mem_size;   /* total initial memory
allocated */
   u_char            *params,    /* parameter pointer */
                    *mem_end;   /* top of memory pointer */
   Mh_exec          mod_head;   /* primary (forked) module
ptr*/
} fork_params, *Fork_params;
```

The child and parent processes execute concurrently. If the parent executes F_WAIT immediately after the fork, it waits until the child dies before it resumes execution. A child process descriptor is returned to free memory only when the parent performs an F_WAIT or an F_EXIT service request.

Modules owned by a super user can execute in system state if the system-state bit in the module's attributes is set. This should only be done when necessary because this process is not time sliced and system protection is not enabled for this process.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`priority` specifies the priority of the new process. If `priority` is zero, the new process inherits the same priority as the calling process.

`path_cnt` specifies the number of I/O paths for the child to inherit.

`proc_id` is a returned value. It is the child process ID.

`mod_name` points to the module name.

`params` points to the parameter string to pass to the new process.

`mem_size` specifies any additional stack space to allocate above the default specified in the primary module's module header.

`param_size` specifies the size of the parameter string.

`type_lang` specifies the desired type and language. If `type_lang` is zero, any module, regardless of type and language, may be loaded.

`orphan` If the `orphan` flag is non-zero, the new process executes without a parent. If `orphan` is zero, the new process is the child of the calling process.

Possible Errors

EOS_NORAM
EOS_PERMIT
EOS_PNNF

See Also

F_CHAIN
F_EXIT
F_WAIT

F_FORKM

Create New Process by Module Pointer

Headers

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

Parameter Block Structure

```
typedef struct f_forkm_pb {
   syscb cb;
   u_int16 priority,
      path_cnt;
   process_id proc_id;
   Mh_com mod_head;
   u_char *params;
   u_int32 mem_size,
      param_size;
   u_int16 orphan;
} f_forkm_pb, *F_forkm_pb;
```

Description

`F_FORKM` 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 new process is forked by a module pointer. `F_FORKM` assumes the module pointer is the primary module pointer for the new process.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`
is the control block header.

`priority`
specifies the priority of the new process. If `priority` is zero, the new process inherits the same priority as the calling process.

`path_cnt`
specifies the number of I/O paths for the child to inherit.

`proc_id` is a returned value. It is the child process ID.

`mod_head` points to the module header of the module to fork.

`params` points to the parameter string to pass to the new process.

`mem_size` specifies any additional stack space to allocate above the default specified in the primary module's module header.

`param_size` specifies the size of the parameter string.

`orphan` If the `orphan` flag is non-zero, the new process executes without a parent. If `orphan` is zero, the new process is the child of the calling process.

Possible Errors

```
EOS_MNF
EOS_NORAM
EOS_PERMIT
```

See Also

`F_FORK`

F_GBLKMP

Get Free Memory Block Map

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_gblkmp_pb {
   syscb cb;
   Mem_list start;
   u_char *buffer;
   u_int32 size,
      min_alloc,
      num_segs,
      tot_mem,
      free_mem;
} f_gblkmp_pb, *F_gblkmp_pb;
```

Description

F_GBLKMP copies the address and size of the system's free RAM blocks into your buffer for inspection. It also returns information concerning the free RAM as noted by the parameters.

A series of structures showing the address and size of free RAM blocks is returned in your buffer in the following format:

```
typedef struct {
     u_char     *address;    /* pointer to block */
     u_int32     size;       /* size of block */
};
```

Although F_GBLKMP returns the address and size of the system's free memory blocks, you cannot directly access these blocks. Use F_SRQMEM to request free memory blocks.

The address and size of free RAM changes with system use. mfree and similar utilities use F_GBLKMP to determine the status of free system memory.

The OS suffixes the array of info structures, to which buffer points, with a sentinel as follows:

info.address	NULL
info.size	0

The OS adds this sentinel only if at least one unused `info` structure occupies the buffer after processing.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`start` is the address to begin reporting the segments.

`buffer` points to the buffer to use.

`size` specifies the buffer size in bytes. It is also an output containing the number of unused `info` structures in the buffer.

When size is `0`, the service does not validate or use `buffer`. It also updates the following parameters on every call.

`min_alloc` is a returned value. It is the minimum memory allocation size for the system.

`num_segs` is a returned value. It is the number of memory fragments in the system.

`tot_mem` is a returned value. It is the total RAM found by the system at startup.

`free_mem` is a returned value. It is the current total free RAM available.

See Also

`F_SRQMEM`

F_GETMDP

Get Current and Alternate Module Directory Pathlists

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_get_mdp_pb {
   syscb cb;
   u_char *current,
      *alternate;
} f_get_mdp_pb, *F_get_mdp_pb;
```

Description

F_GETMDP returns pathlists to the current module directory and the alternate module directory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

current
points to the buffer for returning the pathlist of the current module directory.

alternate
points to the buffer for returning the pathlist of the alternate module directory.

See Also

F_ALTMDIR
F_CHMDIR

F_GETSYS

Examine System Global Variable

Headers

```
#include <types.h>
#include <sysglob.h>
```

Parameter Block Structure

```
typedef struct f_getsys_pb {
   syscb cb;
   u_int32 offset,
         size;
   union {
      u_char byt;
      u_int16 wrd;
      u_int32 lng;
   } sysvar;
} f_getsys_pb, *F_getsys_pb;
```

Description

`F_GETSYS` enables a process to examine a system global variable. Consult the `sysglob.h` header file for a description of the system global variables.

The format and contents of the system global variables may change in future releases of OS-9.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`offset`	is the variable’s offset in the system globals.
`size`	specifies the size of the variable.
`sysvar`	is a union of the three sizes of variables accessible by `F_GETSYS`.

`byt` is a byte size variable.

`wrd` is a word size variable.

`lng` is a long size variable.

See Also

F_SETSYS

The `DEFS` files section of the *OS-9 Porting Guide*

F_GMODDR

Get Copy of Module Directory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_get_moddir_pb {
   syscb cb;
   u_char *buffer;
   u_iont32 count;
} f_get_moddir_pb, *F_get_moddir_pb;
```

Description

F_GMODDR copies the process' current module directory into your buffer for inspection.

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. Therefore, you should use the output of mdir to determine the names of modules in memory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
is the control block header.

buffer
points to the buffer.

count
is the maximum number of bytes to copy, and upon return from F_GMODDR it is the number of bytes actually copied.

Although the module directory contains pointers to each module in the system, never access the modules directly. Instead, use F_CPYMEM to copy portions of the system's address space for inspection.

See Also

`F_CPYMEM`

F_GPRDBT

Get Copy of Process Descriptor Block Table

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_get_prtbl_pb {
   syscb cb;
   u_char *buffer;
   u_int32 count;
} f_get_prtbl_pb, *F_get_prtbl_pb;
```

Description

F_GPRDBT copies the process descriptor block table into your buffer for inspection. The procs utility uses F_GPRDBT to determine which processes are active in the system.

Attributes

Operating System:	OS-9
State:	User and Interrupt
Threads:	Safe

Parameters

cb
is the control block header.

buffer
points to the buffer.

count
is the maximum number of bytes to copy and upon return from F_GPRDBT it is the number of bytes actually copied.

Although F_GPRDBT returns pointers to all process descriptors, never access the process descriptors directly. Instead, use F_GPRDSC to inspect specific process descriptors.

See Also

F_GPRDSC

F_GPRDSC

Get Process Descriptor Copy

Headers

```
#include <process.h>
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_gprdsc_pb {
     syscb       cb;
     process_id  proc_id;
     u_char      *buffer;
     u_int32     count;
     u_int32     type;
} f_gprdsc_pb, *F_gprdsc_pb;
```

Description

F_GPRDSC copies the contents of a process descriptor into the specified buffer for inspection. The procs utility uses F_GPRDSC to obtain information about an existing process.

The format and contents of a process descriptor can change in future releases of OS-9.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

cb	is the control block header.
procid	is the requested process ID.
buffer	points to the buffer.
count	is the maximum number of bytes to copy, and upon return from F_GPRDSC, it is the number of bytes actually copied.

`type`	the type of descritpor to get. The valid values for `type` are `_OS_GET_PRDESC` or `_OS_GET_PRSRC`. `_OS_GET_PRDESC` returns the specified state descriptor and `_OS_GET_PRSRC` returns the specified resource descriptor.

Possible Errors

`EOS_IPRCID`

F_ICPT

Set Up Signal Intercept Trap

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_intercept_pb {
   syscb cb;
   u_int32 (*function)();
   void *data_ptr;
} f_intercept_pb, *F_intercept_pb;
```

Description

F_ICPT tells OS-9 to install a signal intercept routine.

When a process executing an F_ICPT call receives a signal, the process' intercept routine is executed, and the signal code is passed as a parameter. A signal aborts a process that has not used F_ICPT. 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 can be sent at any time, similar to an interrupt. The signal code is passed as a parameter. The intercept routine should be short and fast, such as setting a flag in the process' data area. You should avoid complicated system calls (such as I/O). After the intercept routine has been completed, it may return to normal process execution by executing F_RTE.

Each time the intercept routine is called, the state of the processor (such as its registers) is pushed on to the user's system stack.

Attributes

Operating System:	OS-9
State:	User and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`function` points to the intercept routine.

`data_ptr` points to the intercept routine’s global storage. It usually contains the address of the program’s data area. The syntax for the signal handler is as follows:

```
void usr_sighand(sig_code, sig_count)
signal_code  sig_code;    /* signal received */
u_int32      sig_count;   /* number of signals pending */
```

See Also

F_RTE
F_SEND

F_ID

Get Process ID and User ID

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_id_pb {
   syscb cb;
   process_id proc_id;
   u_int16 priority,
      age;
   int32 schedule;
   owner_id user_id;
} f_id_pb, *F_id_pb;
```

Description

F_ID returns the caller’s process ID number, current process priority and age, scheduling constant, and owner ID. OS-9 assigns the process ID, and each process has a unique process ID. The owner ID is defined in the system password file and is used for system and file security. Several processes can have the same owner ID.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	is a returned value. It is the current process ID number.
priority	is a returned value. It is the priority of the current process.
age	is a returned value. It is the age of the current process.

`schedule` is a returned value. It is the scheduling constant of the current process.

`group` is a returned value. It is the group number of the current process.

`user` is a returned value. It is the user number of the current process.

Possible Errors

`EOS_BPADDR`

F_INITDATA

Initialize Static Storage from Module

Headers

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

Parameter Block Structure

```
typedef struct f_init_data_pb {
   syscb cb;
   Mh_com mod_head;
   u_char *data;
} f_init_data_pb, *F_init_data_pb;
```

Description

F_INITDATA clears the uninitialized data area, copies the module header’s initialized data to the specified data area, and clears the remote data area (if it exists). Next, it adjusts the code and data offsets.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
mod_head	points to the module header.
data	points to the data area.

Possible Errors

EOS_BMHP
EOS_BMID

F_IRQ

Add or Remove Device from IRQ Table

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_irq_pb {
   syscb cb;
   u_int16 vector,
      priority;
   void *irq_entry;
   u_char *statics;
}  f_irq_pb, *F_irq_pb;
```

Description

`F_IRQ` installs an IRQ service routine into the system polling table.

OS-9 does not poll the I/O port prior to calling the interrupt service routine. Device drivers are required to determine if their device caused an interrupt.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb`	is the control block header.
`vector`	specifies the vector number of the associated interrupt.

`priority` specifies the priority. (65535 is reserved.) IRQ service routines for the same vector are placed into a polling table for the vector according to their relative priorities:

- If `priority` is 0, only this device can use the vector.
- If `priority` is 1, it is polled first and no other device can have a priority of one on the vector.
- If `priority` is 65534, it is polled last on the vector.

`irq_entry` points to the IRQ service routine entry point. If `irq_entry` is zero, the call deletes the IRQ service routine.

`statics` points to the global static storage. `statics` must be unique to the device.

Possible Errors

`EOS_POLL` is returned if the polling table is full.

`EOS_PARAM` is returned if an attempt is made to delete an IRQ routine that is not installed for that interrupt.

F_LINK

Link to Memory Module

Headers

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

Parameter Block Structure

```
typedef struct f_link_pb {
   syscb cb;
   u_char *mod_name;
   Mh_com mod_head;
   void *mod_exec;
   u_int16 type_lang,
      attr_rev;
} f_link_pb, *F_link_pb;
```

Description

`F_LINK` searches the current and alternate module directories for a module whose name, type, and language match the parameters.

The module's link count keeps track of how many processes are using the module. If the requested module 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. `F_LINK` cannot find a module whose header has been destroyed (altered or corrupted).

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`
is the control block header.

`mod_name`
points to the module name. If `mod_name` is an explicit module directory pathlist (for example, `/usr/tony/prog`), the `mod_name` pointer is updated to point to the module that was successfully linked (for example, `prog`).

`mod_head`
is a returned value. It is the address of the module's header.

`mod_exec`
is a returned value. It is the pointer to the absolute address of the module's execution entry point. The module header includes this information.

`type_lang`
is the type and language of the module. If `type_lang` is zero, any module can be linked to, regardless of the type and language. Upon completion, `type_lang` is updated with the type/language value from the module's module header.

`attr_rev`
is a returned value. It points to the attribute and revision level of the module.

Possible Errors

EOS_BNAM
EOS_MNF
EOS_MODBSY

See Also

F_LINKM
F_LOAD
F_UNLINK
F_UNLOAD

F_LINKM

Link to Memory Module by Module Pointer

Headers

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

Parameter Block Structure

```
typedef struct f_linkm_pb {
   syscb cb;
   Mh_com mod_head;
   void *mod_exec;
   u_int16 type_lang,
      attr_rev;
} f_linkm_pb, *F_linkm_pb;
```

Description

`F_LINKM` causes OS-9 to link to the module specified by `mod_head`.

The module's link count keeps track of how many processes are using the module. If the requested module is not re-entrant, only one process can link to it at a time.

If the module's access word does not give the process read permission, the link call fails. Link cannot find a module whose header has been destroyed (altered or corrupted).

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`
is the control block header.

`mod_head`
points to the module.

`mod_exec`
is a returned value. It points to the pointer to the absolute address of the module's execution entry point.

`type_lang` is the type and language of the module. If `type_lang` is zero, any module can be linked to regardless of the type and language. Upon completion, `type_lang` is updated with the type/language value from the module's module header.

`attr_rev` is a returned value. It is the attribute and revision level of the module.

Possible Errors

EOS_BNAM
EOS_MNF
EOS_MODBSY

See Also

F_LINK
F_LOAD
F_UNLINK
F_UNLOAD

F_LOAD

Load Module(s) from File

Headers

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

Parameter Block Structure

```
typedef struct f_load_pb {
   syscb cb;
   u_char *mod_name;
   Mh_com mod_head;
   void *mod_exec;
   u_int32 mode;
   u_int16 type_lang,
      attr_rev,
      color;
} f_load_pb, *F_load_pb;
```

Description

F_LOAD loads an OS-9 memory module from a disk file or a serial device (SCF) into the current module directory. When loading from a disk file as specified by mod_name pathlist, the target file is opened and one or more memory modules are read from the file into memory until an error or end of file is reached. When loading from a serial device (SCF), the Kernel attempts to load only one memory module by first reading the header of the module and then the body of the module. In either case, the path to the disk file or serial device is closed after the loading operation.

An error can indicate an actual I/O error, a module with a bad parity or CRC, or insufficient memory of the desired type.

When a module is loaded, its name is added to the calling process' current 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) with a proper module header and CRC. If the file's access mode is S_IEXEC, the file is loaded from the current execution directory. If the file's access mode is S_IREAD, the file is loaded from the current data directory.

If any of the modules loaded belong to the super user, the file must also belong to the super user. This prevents normal users from executing privileged service requests.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`mod_name` points to the module name (pathlist or serial device name).

`mod_head` is a returned value. It is the pointer to the module.

`mod_exec` is a returned value. It is the pointer to the module execution entry point.

`mode` specifies the access mode. The access modes are defined in the `module.h` header file.

`type_lang` is a returned value. It is the type and language of the first module loaded.

`attr_rev` is a returned value. It is the attribute and revision level of the module.

`color` specifies the type of memory in which to load the modules. Modules are loaded into the highest physical memory available of the specified type.

Possible Errors

`EOS_MEMFUL`
`EOS_BMID`

F_MKMDIR

Create New Module Directory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_makmdir_pb {
   syscb cb;
   u_char *name;
   u_int16 perm;
} f_makmdir_pb, *F_makmdir_pb;
```

Description

F_MKMDIR creates a new module directory. The name of the new module directory is relative to the current module directory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
name	points to the name of the new module directory.
perm	specifies the access permissions for the new module directory.

Possible Errors

```
EOS_KWNMOD
EOS_NORAM
```

F_MEM

Resize Data Memory Area

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_mem_pb {
   syscb cb;
   u_char *mem_ptr;
   u_int32 size;
} f_mem_pb, *F_mem_pb;
```

Description

F_MEM contracts or expands the process' data memory area. The size requested is rounded up to an even memory allocation block. Additional memory is allocated contiguously upward (towards higher addresses), or deallocated downward from the old highest address.

This request cannot return all of a process' memory or deallocate the memory at its current stack pointer.

If there is adequate free memory for an expansion request, but the memory is not contiguous, F_MEM returns an error. Memory requests by other processes may have fragmented memory resulting in small, scattered blocks that are not adjacent to the caller's present data area.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`mem_ptr` is a returned value. It is the new end of data segment plus 1.

`size` is the desired memory size in bytes. The actual size of the memory is returned in `size`. If `size` is zero, `F_MEM` treats the call as a request for information and returns the current upper bound in `mem_ptr` and the amount of free memory in `size`.

Possible Errors

EOS_DELSP
EOS_MEMFUL
EOS_NORAM

F_MODADDR

Find Module Given Pointer

Headers

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

Parameter Block Structure

```
typedef struct f_modaddr_pb {
   syscb cb;
   u_char *mem_ptr;
   Mh_com mod_head;
} f_modaddr_pb, *F_modaddr_pb;
```

Description

F_MODADDR locates a module given a pointer to any position with the module and returns a pointer to the module's header.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
mem_ptr	points to any position within the module.
mod_head	is a returned value. It is the pointer to the associated module header.

Possible Errors

EOS_MNF

F_MOVE

Move Data (Low Bound First)

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_move_pb {
   syscb cb;
   u_char *from,
      *to;
   u_int32 count;
} f_move_pb, *F_move_pb;
```

Description

`F_MOVE` is a fast *block-move* subroutine that copies 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 data loss due to incorrect propagation.

Attributes

Operating System:	OS-9
State:	System and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`from`	points to the source data.
`to`	points to the destination data.
`count`	is the byte count to copy.

F_NPROC

Start Next Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_nproc_pb {
   syscb cb;
} f_nproc_pb, *F_nproc_pb;
```

Description

F_NPROC removes the next process from the active process queue and initiates its execution. If there is no process in the queue, OS-9 waits for an interrupt and checks the active process queue again.

F_NPROC does not return to the caller.

The process calling F_NPROC should already be in one of the system's process queues. If not, the process becomes unknown to the system. This occurs even though the process descriptor still exists and is printed out by a procs command.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb	is the control block header.

See Also

F_APROC

F_PERMIT

Allow Access to Memory Block

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_permit_pb {
   syscb cb;
   process_id pid;
   u_int32 size;
   u_char *mem_ptr;
   u_int16 perm;
} f_permit_pb, *F_permit_pb;
```

Description

F_PERMIT is called when a process allocates memory or links to a module to allow the process to access a block of memory.

F_PERMIT must update SSM (System Security Module) data structures to show a process may access the specified memory in the requested mode. F_PERMIT must also increment the number of links to this memory area in a corresponding block count map to keep track of the number of times the same block(s) has been granted access.

A long word (p_spuimg) is reserved in each process descriptor for use by the SSM code. The SSM may allocate data structures for each process and keep a pointer to these structures in p_spuimg.

Note the following:

- The calling process cannot use this service to permit write-only memory or to permit nothing (set no permissions). This service must be used to permit at least read-only access.
- The only user-state processes that may permit memory are the ones in group zero (super user). All other processes must be system-state processes.
- On systems without SSM, the result of any F_PERMIT call is success, regardless of the process state since all processes have full access rights to the entire memory space. When SSM is not

active, the operating system does not validate any of the arguments for this call.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`pid` is the target process' process identifier.

`size` is the size of the memory area.

`mem_ptr` points to the beginning of the memory area to grant access permissions.

`perm` specifies the permissions to add.

Possible Errors

```
EOS_BPADDR
EOS_DAMAGE
EOS_IPRCID
EOS_NORAM
EOS_PARAM
EOS_PERMIT
```

F_PROTECT

Prevent Access to Memory Block

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_protect_pb {
   syscb cb;
   process_id pid;
   u_int32 size;
   u_char *mem_ptr;
   u_int16 perm;
} f_protect_pb, *F_protect_pb;
```

Description

`F_PROTECT` is called when a process deallocates memory or unlinks a module to remove a process' permission to access a block of memory.

The counts in the block count map corresponding to the memory blocks being protected must be decremented and if any block count becomes zero, the protection image must be updated to prevent access to the corresponding memory by the process.

Note the following:

- If `F_PROTECT` is called for a process being debugged, the protection maps of the parent process must also be updated to remove access to the allocated memory.
- The only user-state processes that may protect memory are the ones in group zero (super user). All other processes must be system-state processes.
- On systems without SSM, the result of any `F_PROTECT` call is success, regardless of the process state since all processes have full access rights to the entire memory space. When SSM is not active, the operating system does not validate any of the arguments for this call.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`pid` specifies the process identifier for the target process.

`size` is the size of the memory area.

`mem_ptr` points to the beginning of the memory area to protect access permissions. `size` specifies the size of memory.

`perm` specifies the permissions to remove.

Possible Errors

EOS_BPADDR
EOS_IPRCID
EOS_NORAM
EOS_PERMIT

See Also

F_ALLTSK
F_PERMIT

F_PRSNAM

Parse Path Name

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_prsnam_pb {
   syscb cb;
   u_char *name;
   u_int32 length;
   u_char delimiter,
      *updated;
} f_prsnam_pb, *F_prsnam_pb;
```

Description

`F_PRSNAM` parses a string for a valid pathlist element and returns its size. This call parses one element in a pathname, not the entire pathname. A valid pathlist element may contain the following characters:

`A - Z`Upper case letters`.`Periods

`a - z`Lower case letters _ Underscores

`0 - 9`Numbers $Dollar signs

Other characters terminate the name and are returned as the pathlist delimiter.

Several `F_PRSNAM` calls are needed to process a pathlist with more than one name. `F_PRSNAM` terminates a name when it detects a delimiter character. Usually, pathlists must be terminated with a null byte.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`name` points to the name string.

`length` is a returned value. It is the length of the pathlist element.

`delimiter` is a returned value. It is the pathlist delimiter.

`updated` is a returned value. It is a the pointer to the first character of `name`.

Possible Errors

`EOS_BNAM`

See Also

`F_CMPNAM`

F_RELLK

Release Ownership of Resource Lock

Headers

```
#include <lock.h>
```

Parameter Block Structure

```
typedef struct f_rellk_pb {
   syscb cb;
   lock_id lid;
} f_rellk_pb, *F_rellk_pb;
```

Description

F_RELLK releases ownership of a resource lock and activates the next process waiting to acquire the lock. The next process in the lock's queue is activated and granted exclusive ownership of the resource lock. If no other process is waiting on the lock, the lock is simply marked free for acquisition.

Refer to Chapter 6 for more information about resource locks.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb` is the control block header.

`lid` is the lock identifier of the lock to release.

Possible Errors

`EOS_LOCKID`

See Also

F_ACQLK
F_CAQLK
F_CRLK
F_DELLK
F_WAITLK

F_RETPD

Deallocate Process Descriptor

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_rtnprc_pb {
   syscb cb;
   process_id proc_id;
} f_rtnprc_pb, *F_rtnprc_pb;
```

Description

F_RETPD deallocates a process descriptor previously allocated by F_ALLPRC. You must ensure the process' system resources are returned prior to calling F_RETPD.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	identifies the process descriptor.

Possible Errors

EOS_IPRCID

See Also

F_ALLPRC

F_RTE

Return from Interrupt Exception

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_rte_pb {
   syscb cb;
   u_int32 mode;
} f_rte_pb, *F_rte_pb;
```

Description

F_RTE terminates a process' signal intercept routine and continues executing the main program. However, if unprocessed signals are pending, the intercept routine is re-executed until the queue of signals is exhausted before returning to the main program.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`
is the control block header.

`mode`
is currently unused, but its value must be `0` for future compatibility.

See Also

F_ICPT

F_SEND

Send Signal to Another Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_send_pb {
   syscb cb;
   process_id proc_id;
   signal_code signal;
} f_send_pb, *F_send_pb;
```

Description

F_SEND sends a signal to a specific process. A process may send the same signal to multiple processes of the same group/user ID by passing 0 as the receiving 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 an effective but dangerous tool to get rid of unwanted background tasks.

If an attempt is made to send a signal to a process with a signal pending, the signal is placed in the process’ FIFO signal queue. If the process is in the signal intercept routine when it receives a signal, the new signal is processed when F_RTE is executed by the target process.

If the destination process for the signal is sleeping or waiting, it is activated to 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).

The wake-up signal is an exception. It activates a sleeping process but does not invoke the signal intercept routine. The wake-up signal does not abort a process that has not made an `F_ICPT` call. Wake-up signals never queue and have no effect on active processes in user state. User programs should avoid using the wake-up signal since it is used by the system to activate sleeping processes. Signal codes are defined as follows:

Code	Value	Description
`S_WAKE`	1	Wake up process
`S_QUIT`	2	Keyboard abort
`S_INT`	3	Keyboard interrupt
`S_KILL`	4	System abort (unconditional)
`S_HANGUP`	5	Hang-up
	6-19	Reserved for future use by Microware (globally definable)
	20-25	Reserved for Microware for specific platforms (locally definable)
	26-31	User definable for specific platforms
	32-127	Reserved for Microware (Ultra C)
	128-191	Reserved for Microware for specific platforms (locally definable)
	192-255	Reserved for Microware (globally definable)
	256- 4294967295	User definable

The `S_KILL` signal may only be sent to processes with the same group ID as the sender. Super users may kill any process.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`proc_id`	is the process ID number for the intended receiver. A `proc_id` of zero specifies all processes with the same group/user ID.
`signal`	specifies the signal code to send.

Possible Errors

`EOS_IPRCID`
`EOS_SIGNAL`
`EOS_USIGP`

See Also

`F_ICPT`
`F_RTE`
`F_SIGMASK`
`F_SLEEP`
`F_WAIT`

F_SETCRC

Generate Valid CRC in Module

Headers

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

Parameter Block Structure

```
typedef struct f_setcrc_pb {
   syscb cb;
   Mh_com mod_head;
} f_setcrc_pb, *F_setcrc_pb;
```

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 that is subsequently written to a file. The module must have the correct size and sync bytes; other parts of the module are not checked.

The module image must start on a longword address or an exception may occur.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
mod_head	points to the module.

Possible Errors

EOS_BMID

See Also

F_CRC

F_SETSYS

Set or Examine OS-9 System Global Variables

Headers

```
#include <sysglob.h>
```

Parameter Block Structure

```
typedef struct f_setsys_pb {
  syscb cb;
  u_int32 offset,
    size;
  union {
  u_char byt;
  u_int16 wrd;
  u_int32 lng;
  } sysvar;
} f_setsys_pb, *F_setsys_pb;
```

Description

F_SETSYS changes or examines a system global variable.   These variables have a d_ prefix in the system header file library sysglob.h. Consult the DEFS files for a description of the system global variables.

Only super users may change system variables. You can examine and change any system variable, but typically, only d_minpty and d_maxage are changed. Consult Chapter 1 for an explanation of these variables.

Super users must be extremely careful when changing system global variables.

The system global variables are OS-9's data area. They are highly likely to change from one release to another. You may need to relink programs using this system call to be able to run on future versions of OS-9.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`offset` is the offset to the system globals.

`size` specifies the size of the target variable and which union variable is to be used to set the target global variable.

`sysvar` is a union of the three sizes of variables accessible by `F_SETSYS`.

`byt` is the byte size variable.

`wrd` is the word size variable.

`lng` is the long size variable.

EXAMPLE

```
#include <sysglob.h>
u_int16 min_priority;

_os_setsys(OFFSET(Sysglobs, d_minpty),
sizeof(u_int16),&min_priority);
```

Possible Errors

```
EOS_PARAM
EOS_PERMIT
```

See Also

`F_GETSYS`

F_SIGLNGJ

Set Signal Mask Value and Return on Specified Stack Image

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_siglngj_pb {
   syscb cb;
   void *usp;
   u_int16 siglvl;
} f_siglngj_pb, *F_siglngj_pb;
```

Description

`F_SIGLNGJ` allows processes to perform `longjump()` operations from their signal intercept routines and unmask signals in one operation.

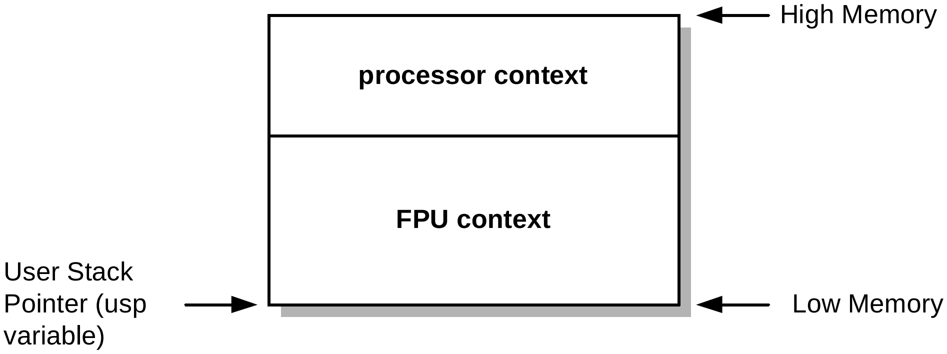
This call is usually used by nested intercept routines to resume execution in the process at a different location from where the process was interrupted by the original signal. When this call is made, the operating system performs the following functions:

- Validates and copies the target process stack image from the memory buffer pointed to by the `usp` variable to the process' system state stack
- Sets the process' signal mask to the value specified in the `siglvl` variable
- Returns to the process restoring the context copied from the user state process stack image

The operating system takes appropriate precautions to verify the memory location pointed to by the `usp` variable is accessible to the process and to ensure the process does not attempt to make a state change.

The stack image pointed to by the usp variable must have the format shown in <Bold><links>Figure 0-1.

Figure 0-1 `F_SIGNLNGJ` **Required Stack Image**



The specific format of the processor context is defined by the `longstk` structure definition found in the `reg<CPU Family>.h` file for the associated processor. The format of the floating-point context varies depending on whether the target system has a hardware floating-point unit versus floating-point emulation software.

For floating-point hardware, the stack image is the same as that defined by the `fregs` structure definition found in the associated `reg<CPU Family>.h` header file.

For floating-point emulation, the floating-point context differs from the hardware implementation context as it may contain additional context information specific to the FPU module performing the emulation. The definition for the floating-point context as used by the FPU module is the `fpu_context` structure defined in the associated `reg<CPU Family>.h` header file for the target processor.

If a particular application needs to access the contents of the process context, it may use the size of these structures for indexing. Alternatively, the application can determine the size of the FPU context at runtime by accessing the kernel globals field, `d_fpusize`, containing the size of the FPU context.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

`cb` is the control block header.

`usp` points to the new process stack image.

`siglvl` is the new signal level value.

Possible Errors

EOS_PARAM

See Also

F_SEND
F_SIGMASK
F_SLEEP
F_WAIT

F_SIGMASK

Mask or Unmask Signals During Critical Code

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_sigmask_pb {
   syscb cb;
   u_int32 mode;
} f_sigmask_pb, *F_sigmask_pb;
```

Description

F_SIGMASK enables signals to reach the calling process or disables signals from reaching the calling process. If a signal is sent to a process whose mask is non-zero, the signal is queued until the process mask becomes zero. The process' signal intercept routine is executed with signals inherently masked. New processes begin with a signal mask value of zero (not 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.

By doing additions and subtractions (instead of merely just setting a flag), this service allows the OS and the process in question to nest the masking and unmasking of multiple signals. Also, since a process may want to receive signals without nesting back out through a bunch of F_SIGMASK calls, the OS provides three ways for clearing the mask (i.e., nesting level): F_SIGMASK with a "mode" argument of zero, F_SLEEP, and F_WAIT.

This service returns the EOS_PARAM error code whenever the calling process specifies a "mode" argument other than negative one, zero, or one (i.e., -1, 0, or 1).

Signals are analogous to hardware interrupts and should be masked sparingly. Keep intercept routines as short and fast as possible.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`mode`	is the process signal level.

Mode	Description
0	The signal mask is cleared.
1	The signal mask is incremented.
-1	The signal mask is decremented.

Possible Errors

`EOS_PARAM`

See Also

`F_SEND`
`F_SLEEP`
`F_WAIT`

F_SIGRESET

Reset Process Intercept Routine Recursion Depth

Headers

```
#include <signal.h>
```

Parameter Block Structure

```
typedef struct f_sigrst_pb {
   syscb cb;
} f_sigrst_pb, *F_sigrst_pb;
```

Description

F_SIGRESET should be used whenever a program uses a longjmp() to get out of an intercept routine or unmasks signals in an intercept service routine with the intent of never using the F_RTE call to return.

```
   if(setjmp(x) != 0) {
     _os_sigreset();
     _os_sigmask(-1);
   }
```

Under normal circumstances, OS-9 keeps the state of the main process on the system stack while a signal intercept routine executes. However, if the 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 uses a longjmp() call 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_RTE calls, the states must be left where they are.

If the code uses a longjmp() call 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 through a F_SIGRESET call.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

`cb`	is the control block header.

See Also

`F_ICPT`
`F_RTE`

F_SIGRS

Resize Process Queue Block Parameter Block

Headers

```
#include <srvcb.h>
```

Parameter Block Structure

```
typedef struct f_sigrs_pb {
   syscb cb;
   u_int32 signals;
} f_sigrs_pb, *F_sigrs_pb;
```

Description

`F_SIGRS` allows a process to change the maximum number of signals queued on its behalf.

You can use this call to increase or decrease the number of signals queued. An error is returned (`EOS_PARAM`) if a request is made to reduce the number of queued signals while there are signals pending. The initial default for the system is specified in the system `init` module.

This service returns `EOS_PARAM` if the user requests a signal-queue size of zero (while the OS has no signals pending for this process) or a signal-queue size less than the number of maximum signals (e.g., trying to resize the queue to hold only five signals when the OS has one signal pending for a process whose maximum signal count is ten).

This service returns `EOS_NORAM` if the process requests a queue whose size is larger than available memory.

This service does not allow the caller to set the queue's size to zero. However, the caller (if and only if there are no signals pending) can use this service to decrease the size of the queue (even down to one). If there are pending signals, however, then the value for `signals` must be greater than or equal to the maximum number of signals that the process' queue can hold.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`signals`	is the new maximum number of signals.

Possible Errors

```
EOS_PARAM
EOS_NORAM
EOS_DAMAGE
```

See Also

`F_SIGRESET`

F_SLEEP

Put Calling Process to Sleep

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_sleep_pb {
   syscb cb;
   u_int32 ticks;
   signal_code signal;
} f_sleep_pb, *F_sleep_pb;
```

Description

F_SLEEP deactivates the calling process until the requested number of ticks have elapsed.

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 request to surrender the current time slice. 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 inserts the process in 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 (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 and may be determined using F_TIME system call. If the high order bit of the *ticks* parameter is set, the low 31 bits are interpreted as 1/256 second and converted to ticks before sleeping. This allows program delays to be independent of the system’s clock rate.

This function does not return any error code if the operating system cannot wait for the requested time due to an overflow when converting a time from 256ths-of-a-second into clock ticks. This only occurs if you specify a time in 256ths-of-a-second and the system clock ticks occur at a rate greater than 512 ticks-per-second. If an overflow occurs, the operating system waits for the longest delay possible.

The system clock must be running to perform a timed sleep. The system clock is not required to perform an indefinite sleep or to give up a time slice.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`ticks` is the length of time to sleep in ticks/second.

- If `ticks` is zero, the process sleeps indefinitely.
- If `ticks` is one, the process gives up a time slice but does not necessarily sleep for one tick.

`signal` is a returned value. It is the last signal the process received. `signal` is returned to the calling process at the completion of the sleep.

- If `signal` is zero, the process slept for the time specified by ticks.
- If `signal` is non-zero, the number corresponds to the signal that awoke the process.

Possible Errors

EOS_NOCLK

See Also

F_SEND
F_TIME
F_WAIT

F_SLINK

Install User Subroutine Module

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_slink_pb {
   syscb cb;
   u_int16 sub_num;
   u_char *mod_name;
   void *lib_exec;
   u_char *mem_ptr;
   Mh_com *mod_head;
} f_slink_pb, *F_slink_pb;
```

Description

Subroutine libraries provide a convenient way to link to a standard set of routines at execution time. Use of subroutine libraries keeps user programs small and automatically updates programs using the subroutine code if it is changed. This is accomplished without recompiling or relinking the program itself. Most Microware utilities use one or more subroutine libraries.

`F_SLINK` attempts to link or load the named module. It returns a pointer to the execution entry point and a pointer to the library's static data area for subsequent calls to the subroutine.The calling program must store and maintain the subroutine's entry point and data pointer. The calling program must also set the subroutine library's data pointer and dispatch to the correct address.

You can remove a subroutine by passing a null pointer for the name of the module and specifying the subroutine number. A process can link to a maximum of 16 subroutine libraries, numbered from 0 to 15.

The return value in the case of an error is -1, even though the type is a pointer and a null is more common.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`sub_num` is the subroutine number.

`mod_num` points to the name of the subroutine module.

`lib_exec` is a returned value. It points to the subroutine entry point.

`mem_ptr` is a returned value. It points to the subroutine static memory.

`mod_head` is a returned value. It points to the module header.

Possible Errors

```
EOS_BPNAM
EOS_ISUB
EOS_NORAM
EOS_PERMIT
```

See Also

`F_TLINK`

F_SLINKM

Link to Subroutine Module by Module Pointer

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_slinkm_pb {
   syscb cb;
   u_int16 sub_num;
   Mh_com mod_head;
   void *lib_exec;
   u_char *mem_ptr;
} f_slinkm_pb, *F_slinkm_pb;
```

Description

`F_SLINKM` is passed a pointer to the subroutine library module to install. If a subroutine library already exists for the specified subroutine number, an error is returned. If static storage is required for the subroutine library, it is allocated and initialized.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`sub_num`	is the subroutine number.
`mod_head`	points to the module header.
`lib_exec`	is a returned value. It points to the subroutine entry point.
`mem_ptr`	is a returned value. It points to the subroutine static memory.

Possible Errors

EOS_NORAM
EOS_PERMIT

See Also

F_TLINKM

F_SPRIOR

Set Process Priority

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_setpr_pb {
   syscb cb;
   process_id proc_id;
   u_int16 priority;
} f_setpr_pb, *F_setpr_pb;
```

Description

F_SPRIOR changes the process priority to the value specified by priority. A super user (group ID zero) may change any process' priority. A non-super user can only change the priorities of processes with the same user ID.

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.

These variables are initially set in the Init module.

A small change in relative priorities has a tremendous effect. For example, if two processes have the 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.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` is the process ID.

`priority` specifies the new priority. 65535 is the highest priority; 0 is the lowest.

Possible Errors

`EOS_IPRCID`

F_SRQMEM

System Memory Request

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_srqmem_pb {
    syscb cb;
    u_char *mem_ptr;
    u_int32 size;
    u_int16 color;
} f_srqmem_pb, *F_srqmem_pb;
```

Description

F_SRQMEM allocates a block of a specific type of memory.

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).

The requested number of bytes is rounded up to a system defined blocksize (currently 16 bytes). F_SRQMEM is useful for allocating I/O buffers and any other semi-permanent memory. The memory always begins on an even boundary.

Memory types or *color codes* are system dependent and may be arbitrarily assigned by the system administrator. Microware reserves values below 256 for future use.

Do not use F_SRQMEM from Interrupt Service Routines.

The byte count of allocated memory and the pointer to the block allocated must be saved if the memory is ever to be returned to the system.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`mem_ptr` points to the allocated memory block.

`size` specifies the byte count of the requested memory. If `size` is `-1`, the largest block of free memory of the specified type is allocated to the calling process. Upon completion of the service request, `size` contains the actual size of the memory block allocated.

`color` specifies the memory type.

- If `color` is non-zero, the search is restricted to memory areas of that color. The area with the highest priority is searched first.
- If `color` is zero, the search is based only on priority. This allows you to configure a system such that fast on-board memory is allocated before slow off-board memory. Areas with a priority of zero are excluded from the search.

Possible Errors

`EOS_MEMFUL`
`EOS_NORAM`

See Also

`F_SRTMEM`

F_SRTMEM

Return System Memory

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_srtmem_pb {
   syscb cb;
   u_char *mem_ptr;
   u_int32 size;
} f_srtmem_pb, *F_srtmem_pb;
```

Description

F_SRTMEM deallocates memory when it is no longer needed. The returned number of bytes is rounded up to a system defined blocksize before returning the memory. Rounding occurs identically to that performed by F_SRQMEM.

In user state, the system keeps track of memory allocated to a process and all blocks not returned are automatically deallocated by the system when a process terminates.

In system state, the process must explicitly return its memory.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
mem_ptr	points to the memory block to return.
size	specifies the byte count of the returned memory.

Possible Errors

EOS_BPADDR

See Also

F_MEM
F_SRQMEM

F_SSPD

Suspend Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_sspd_pb {
    syscb cb;
    process_id proc_id;
} f_sspd_pb, *F_sspd_pb;
```

Description

F_SSPD temporarily suspends a process. A super user (group ID zero) may suspend any process. A non-super user can only suspend processes with the same user ID.

The process may be reactivated with F_APROC.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
proc_id	identifies the target process.

Possible Errors

EOS_IPRCID

See Also

F_APROC

F_SSVC

Service Request Table Initialization

Headers

```
#include <types.h>
#include <svctbl.h>
```

Parameter Block Structure

```
typedef struct f_ssvc_pb {
   syscb cb;
   u_int32 count;
   u_int16 state_flag;
   void *init_tbl,
      *params;
} f_ssvc_pb, *F_ssvc_pb;
```

Description

F_SSVC adds or replaces service requests in OS-9's user and privileged system service request tables.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb`	is the control block header.
`count`	is a count of the entries in the initialization table.

`state_flag` specifies whether user or system state tables, or both, are updated.

- If `state_flag` is `1`, only the user table is updated.
- If `state_flag` is `2`, only the system table is updated.
- If `state_flag` is `3`, both the system and user tables are updated.

`init_tbl` points to the initialization table. An example initialization table might look like this:

```
error_code printmsg(), service();
svctbl  syscalls[] =
{
    {F_PRINT, printmsg},
    {F_SERVICE, service}
};
```

`params` may be a pointer to anything, but is intended to be a pointer to global static storage. Whenever a system call is executed, the `params` data pointer is passed automatically.

The following structure definition of the initialization table is located in `svctbl.h`:

```
#if !defined(_TYPES_H)
#include <types.h>
#endif
#define USER_State    1   /* user-state service routine flag */
#define SYSTEM_State  2   /* system-state service routine flag */
/* service routine initialization table structure. */
typedef struct  {
  u_int16   fcode;        /* system call function code */
  u_int32   (*service)();  /* service routine pointer */
  u_int32   attr;         /* attributes of system call (reserved for
future use) */
  u_int16   ed_low,       /* low bound of acceptable service call
edition */
            ed_high       /* upper bound of edition */
} svctbl, *Svctbl;

#endif
```

F_STIME

Set System Date and Time

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_setime_pb {
   syscb cb;
   u_int32 time;
} f_setime_pb, *F_setime_pb;
```

Description

F_STIME sets the current system time and starts the system real-time clock to produce time slice interrupts. F_STIME puts the time in the system static storage area and links the clock module. If the link is successful, the clock initialization routine is called.

The clock module has three responsibilities:

1. Sets up hardware-dependent functions to produce system tick interrupts. This could include moving the new time into the hardware.
2. Installs a service routine to clear the interrupt when a tick occurs.
3. The interrupt service routine must call through to the kernel's *tick* routine to allow the kernel to keep accurate time in software. The address to the kernel's tick routine is provided by the kernel via the clock module's static storage structure when the kernel initializes the clock module.

The OS-9 kernel keeps track of the current time in software, which enables clock modules to be small and simple. Some OS-9 utilities and functions expect the clock to have the correct time. Therefore, you should run F_STIME whenever the system is started. This is usually done in the system startup file.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`time`	specifies the time. The time is stored as the number of seconds since 1 January 1970 Greenwich Mean Time.

The time is not validated in any way. If `time` is zero on systems with a battery-backed clock, the actual time is read from the real-time clock.

Possible Errors

```
EOS_MNF
EOS_NOCLK
EOS_NORAM
```

See Also

`F_TIME`

F_STRAP

Set Error Trap Handler

Headers

```
#include <types.h>
#include <settrap.h>
#include <regs.h>
```

Parameter Block Structure

```
typedef struct f_strap_pb {
   syscb cb;
   u_int32 *excpt_stack;
   Strap init_tbl;
} f_strap_pb, *F_strap_pb;

typedef struct strap (
   u_int32 vector,
      (*routine)();
} strap, *Strap;
```

Description

F_STRAP enables the user programs to catch program exceptions such as illegal instructions and divide-by-zeroes. The exceptions that may be trapped are processor-dependent.

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.

If a user routine is not provided and one of these exceptions occurs, the program is aborted.

When a user's exception routine is executed, it is passed the following information.

```
void errtrap(vector_errno, badpc, badaddr)
u_int32 vector_errno,  /*error number of the vector */
        badpc,         /* PC where exception occurred */
        badaddr;       /*address where exception
occurred.*/
```

You can disable an error exception handler by calling F_STRAP with an initialization table specifying 0 as the offset to the routine(s) to remove.

For example, the following table would remove user routines for TRAPV and CHK error exceptions.

```
Strap errtab[] = {
   {T_BUSERR, 0},
   {T_ADDERR, 0},
   {-1, NULL}
};
```

Beware of exceptions in exception handling routines. They are usually not re-entrant.

Attributes

Operating System:	OS-9
State:	User
Threads:	Safe

Parameters

`cb` is the control block header.

`excpt_stack` points to the stack to use if an exception occurs. If `excpt_stack` is zero, `F_STRAP` uses the current stack.

`init_tbl` points to the service request initialization table. An initialization table might appear as follows:

```
Strap errtab[] = {
{T_BUSERR, errtrap},
{T_ADDERR, errtrap},
{-1, NULL}
};
```

Possible Errors

EOS_TRAP

See Also

F_ABORT

F_SUSER

Set User ID Number

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_setuid_pb {
   syscb cb;
   owner_id user_id;
} f_setuid_pb, *F_setuid_pb;
```

Description

F_SUSER changes the current user ID to user_id.

The following restrictions apply to F_SUSER:

- Users with group ID zero may change their IDs to anything.
- A primary module owned by a group zero user may change its ID to anything.
- Any primary module may change its user ID to match the module’s owner.

All other attempts to change the user ID number return an EOS_PERMIT error.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
user_id	is the desired group/user ID number.

Possible Errors

EOS_PERMIT

F_SYSDBG

Call System Debugger

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_sysdbg_pb {
   syscb cb;
   void *param1,
      *param2;
} f_sysdbg_pb, *F_sysdbg_pb;
```

Description

F_SYSDBG calls the system level debugger, if one exists. This allows you to debug system-state routines, such as device drivers. The caller defines the parameters to this service request to values useful in debugging. For example, a parameter could be a pointer to a critical data structure.

When the system level debugger is active, it runs in system state and effectively stops timesharing. F_SYSDBG can only be called by users in group zero. Never use this call when other users are on the system.

The break utility calls F_SYSDBG.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb — is the control block header.

param1 and param2 — are parameters passed to the debugger. These are currently not used.

Possible Errors

EOS_PERMIT

F_SYSID

Return System Identification

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_sysid_pb {
   syscb cb;
   u_int32 oem,
      serial,
      mpu_type,
      os_type,
      fpu_type;
   int32 time_zone
   u_int32 resv1,
      resv2;
   u_char *sys_ident,
      *copyright,
      *resv3;
} f_sysid_pb, *F_sysid_pb;
```

Description

`F_SYSID` returns information about the system.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`oem` is the OEM identification number.

`serial` is the copy serial number.

`mpu_type` is the processor identifier (for example 80386).

`os_type` is the kernel (OS) MPU configuration.

`fpu_type` is the floating-point processor identifier (for example 80387).

`time_zone` is the system time zone in minutes offset from Greenwich Mean Time (GMT).

`resv1`, `resv2`, and `resv3` are reserved pointers.

`sys_ident` points to a buffer for the system identification message.

`copyright` points to a buffer for the copyright message.

F_THEXIT

Exit a Thread

Headers

```
#include <threads.h>
```

Parameter Block Structure

```
typedef struct f_thexit_pb {
   syscb          cb;
   error_code     status;
} f_thexit_pb, *F_thexit_pb;
```

Description

F_THEXIT causes the calling thread to exit. If the calling program is not multi-threaded, the EOS_PERMIT error is returned.

If successful, F_THEXIT does not return to the caller.

Threads created via pthread_create() should not use this call. Doing so results in instability and loss of resources for the process.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	System call control block.
status	Exit status.

See Also

F_THFORK

F_THFORK

Fork a Thread

Headers

```
#include <threads.h>
```

Parameter Block Structure

```
typedef struct f_thfork_pb {
   syscb           cb;
   thread_t        thread_id;
   thread_attr_t   attr;
   void            *stack_top;
   void            *start_addr;
   void            *arg;
   void            *data;
   u_int32         stack_size;
} f_thfork_pb, *F_thfork_pb;
```

Description

F_THFORK forks a new thread of control in the current process.

Threads created with _os_thfork() or F_THFORK are not permitted to use C library calls that have threading issues. Create threads with pthread_create().

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb
System call control block.

attr
Input: thread attribute structure.

*stack_top
Input: thread's initial stack pointer.

*start_addr
Input: thread's initial execution address.

*arg
Input: argument passed to thread.

`*data` Input: thread specific data pointer.

`stack_size` Input/output: size of stack to allocate/allocated.

See Also

F_THEXIT
F_THREAD

F_THREAD

Set Thread Parameters

Headers

```
#include <funcs.h>
#include <threads.h>
```

Parameter Block Structure

```
typedef struct f_thread_pb {
     syscb          cb;
     u_int32        code;
     thread_t       thread_id;
     void           *pb;
} f_thread_pb, *F_thread_pb;
```

Description

F_THREAD sets thread parameters for the thread specified by thread_id.

If code is TH_TSDATA, arg is the thread specific data pointer.

Threads created via pthread_create() should not use this call. Doing so results in instability and loss of resources for the process.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	System call control block.
code	Thread action code: TH_ORPHAN, etc.
thread_id	Thread ID if applicable.
*pb	Additional parameters.

See Also

F_THEXIT
F_THFORK

F_TIME

Get System Date and Time

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_getime_pb {
   syscb cb;
   u_int32 time,
      ticks;
} f_getime_pb, *F_getime_pb;
```

Description

F_TIME returns the current system time in the number of seconds since 1 January 1970 Greenwich Mean Time.

F_TIME returns a date and time of zero (with no error) if no previous call to F_STIME has been made. A tick rate of zero indicates the clock is not running.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
time	is a returned value. It is the current time.
ticks	contains the following:

- The clock tick rate in ticks per second is returned in the most significant word.
- The least significant word contains the current tick.

See Also

F_STIME

F_TLINK

Install System State Trap Handling Module

Headers

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

Parameter Block Structure

```
typedef struct f_tlink_pb {
   syscb cb;
   u_int16 trap_num;
   u_char *mod_name;
   void *lib_exec,
      *mod_head,
      *params;
   u_int32 mem_size;
} f_tlink_pb, *F_tlink_pb;
```

Description

Trap handlers enable a program to execute privileged (system state) code without running the entire program in system state. Trap handlers only run in system state.

`F_TLINK` attempts to link or load the module specified by `mod_name`. If the link/load is successful, `F_TLINK` installs a pointer to the module 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. If static storage is required for the trap handler, OS-9 allocates and initializes it.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb` is the control block header.

`trap_num` specifies the user trap number (1 through 15).

`mod_name` points to the name of the trap module. If `mod_name` is zero or points to a null string, the trap handler is unlinked.

`lib_exec` points to the pointer to the trap execution entry point.

`mod_head` points to the pointer to the trap module.

`params` is a reserved field.

`mem_size` specifies the additional memory size to be allocated for the trap modules static data area.

Possible Errors

EOS_ITRAP
EOS_MNF
EOS_NORAM
EOS_PERMIT

See Also

F_SLINK

F_TLINKM

Install User Trap Handling Module by Module Pointer

Headers

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

Parameter Block Structure

```
typedef struct f_tlinkm_pb {
   syscb cb;
   u_int16 trap_num;
   Mh_com mod_head;
   void *lib_exec;
   void *params;
   u_int32 mem_size;
} f_tlinkm_pb, *F_tlinkm_pb;
```

Description:

`F_TLINKM` is passed a pointer to the module to install. If a trap module already exists for the specified trap number, an error is returned. If static storage is required for the trap handler, it is allocated and initialized.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`trap_num`	specifies the user trap number (0 through 15).
`mod_head`	points to the module header.
`lib_exec`	points to the trap execution entry point.
`params`	is a reserved field.
`mem_size`	specifies the additional memory size to be allocated for the trap module’s static data area.

Possible Errors

EOS_ITRAP
EOS_NORAM
EOS_PERMIT

See Also

F_TLINK

F_UACCT

User Accounting

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_uacct_pb {
   syscb cb;
   u_int16 acct_code;
   Pr_desc proc_desc;
} f_uacct_pb, *F_uacct_pb;
```

Description

F_UACCT provides a means for users to set up an accounting system. The kernel calls F_UACCT whenever it forks or exits a process. Therefore, F_UACCT provides a mechanism for users to keep track of system operators.

To install a handler for this service request, use the F_SSVC system call to add the user's accounting routine to the system's service request dispatch table. This is usually done in an OS9P2 module.

You may perform your own system accounting by calling F_UACCT with a user defined acct_code identifying the operation to perform. For example, when the kernel forks a process it identifies the operation by passing the F_FORK code to the accounting routine.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`acct_code` is the operation identifier. This is usually a system call function code.

`proc_desc` points to the current process descriptor.

Possible Errors

EOS_UNKSVC (This error should be ignored.)

See Also

F_SSVC

F_UNLINK

Unlink Module by Address

Headers

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

Parameter Block Structure

```
typedef struct f_unlink_pb {
   syscb cb;
   Mh_com mod_head;
} f_unlink_pb, *F_unlink_pb;
```

Description

F_UNLINK notifies OS-9 the calling process no longer needs a module. The module's link count is decremented. When the link count equals zero (-1 for sticky modules), the module is removed from the module directory and its memory is deallocated. When several modules are loaded together as a group, they are only removed when the link count of all modules in the group are zero (-1 for sticky modules).

Some modules cannot be unlinked; for example, device drivers in use and all modules included in the bootfile.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
mod_head	points to the module header.

Possible Errors

EOS_MODBSY

See Also

F_LINK
F_UNLOAD

F_UNLOAD

Unlink Module by Name

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_unload_pb {
   syscb cb;
   u_char *mod_name;
   u_int16 type_lang;
} f_unload_pb, *F_unload_pb;
```

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. A sticky module is not removed until its link count is -1. This call is similar to F_UNLINK, except F_UNLOAD is passed the pointer to the module name instead of the address of the module header.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	is the control block header.
mod_name	points to the module name.
type_lang	specifies the module's type and language.

Possible Errors

EOS_MNF
EOS_MODBSY

See Also

F_LINK
F_UNLINK

F_VMODUL

Verify Module

Headers

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

Parameter Block Structure

```
typedef struct f_vmodul_pb {
   syscb cb;
   Mh_com mod_head,
      mod_block;
   u_int32 block_size;
} f_vmodul_pb, *F_vmodul_pb;
```

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. The current module directory is 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.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

cb
is the control block header.

mod_head
points to the module.

mod_block
points to the memory block containing the module.

block_size
is the size of the memory block containing the module.

Possible Errors

EOS_BMCRC
EOS_BMHP
EOS_BMID
EOS_DIRFUL
EOS_KWNMOD

See Also

F_CRC
F_LOAD

F_WAIT

Wait for Child Process to Terminate

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_wait_pb {
   syscb cb;
   process_id child_id;
   status_code status;
} f_wait_pb, *F_wait_pb;
```

Description

`F_WAIT` deactivates the calling process until a child process terminates. The child's ID number and exit status are returned to the parent.

If the caller has several children, the caller is activated when the first child dies, so one `F_WAIT` call is required to detect the termination of each child.

If a child died before the `F_WAIT` call, the caller is reactivated immediately. `F_WAIT` returns an error only if the caller has no children.

The process descriptors for child processes are not returned to free memory until their parent process performs an `F_WAIT` system call or terminates.

If a signal is received by a process waiting for children to terminate, the process is activated. In this case, `child_id` contains zero, because no child process has terminated.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`cb`	is the control block header.
`child_id`	is the process ID of the terminating child.
`status`	is the child process' exit status code.

Possible Errors

`EOS_NOCHLD`

See Also

`F_EXIT`
`F_FORK`
`F_SEND`

F_WAITID

Wait for a Specified Process or Thread to Exit

Headers

```
#include <process.h>
```

Parameter Block Structure

```
typedef struct f_waitid_pb {
     syscb           cb;
     process_id      child_id;
     status_code     status;
     signal_code     signal;
     u_int32         wait_flag;
} f_waitid_pb, *F_waitid_pb;
```

Description

F_WAITID has two primary functions:

- waiting for a child process or sibling thread
- controlling a signal for the death of a child process or sibling thread

Waiting for a Child Process or Sibling Thread

To specify a wait related activity, wait_flag should be 0.

child_id specifies the process or thread to wait for. If the value of child_id is the ID of a thread, the caller must be a thread in the same process as child_id. Otherwise an EOS_IPRCID error is returned.

If the call is successful, the exit code of child_id is returned in status.

If the wait is interrupted by a signal, a value of EOS_BSIG is returned by F_WAITID and the value of the signal that caused the interruption is stored in signal.

Controlling a Signal for the Death of a Child Process or Sibling Thread

To specify a signal related activity, wait_flag should be non-zero. The valid values for wait_flag are WT_SIGNAL and WT_RELEASE.

When wait_flag is WT_SIGNAL it specifies that the caller wants to receive a signal when the process or thread specified by child_id terminates. The value of the signal to be sent is signal. If the process or thread specified by child_id has already terminated, the signal is sent immediately.

When wait_flag is WT_RELEASE it specifies that the caller is no longer interested in getting a signal on the termination of the process or thread specified by child_id. signal is irrelevant in this case.

status is not modified when wait_flag is non-zero.

F_WAITID returns immediately when wait_flag is non-zero; it never blocks, regardless of the state of the child.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	System call control block.
child_id	Input: Child process or Thread ID
status	Output: Status of child.
signal	Input: Signal to send.
	Output: Signal that aborted blocking wait.
wait_flag	Input: wait condition flag.

See Also

F_EXIT
F_THEXIT
F_WAIT

F_WAITLK

Activate Next Process Waiting to Acquire Lock

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct f_waitlk_pb {
   syscb cb;
   lock_id lid;
   signal_code signal;
} f_waitlk_pb, *F_waitlk_pb;
```

Description

F_WAITLK activates the next process waiting to acquire the lock. The next process in the lock's queue is activated and granted exclusive ownership of the resource lock. If no other process is waiting on the lock, the lock is simply marked free for acquisition. In either case, the calling process is suspended and inserted into a waiting queue for the resource based on relative scheduling priority.

If, during the course of waiting on a lock, a process receives a signal, the process is activated without gaining ownership of the lock.

The process returns from the wait lock call with an EOS_SIGNAL error code and the signal code is returned via the signal pointer.

If an S_WAKEUP signal is received by a waiting process, the signal code does not register and will be zero.

Refer to Chapter 6 for more information about resource locks.

Attributes

Operating System:	OS-9
State:	System
Threads:	Safe

Parameters

`cb`	is the control block header.
`lid`	is the lock ID on which to wait.
`signal`	points to the received signal.

Possible Errors

`EOS_SIGNAL`

See Also

`F_ACQLK`
`F_CAQLK`
`F_CRLK`
`F_DELLK`
`F_RELLK`

F_YIELD

Yield the Processor

Headers

```
#include <process.h>
```

Parameter Block Schedule

```
typedef struct f_yield_pb {
     syscb     cb;
} f_yield_pb, *F_yield_pb;
```

Description

F_YIELD causes the calling process or thread to be placed back into the active queue. The active queue contents are aged and the highest aged process is given control of the processor. In other words, F_YIELD causes the operating system to advance to the next executable process or thread. It is possible that the next executable process or thread will be the one that called F_YIELD. The status of the process' or thread's signal mask remains unchanged during this system call. F_YIELD is much like F_SLEEP with a tick count of 1, except that signals are not implicitly unmasked.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

cb	System call control block.

See Also

F_SLEEP

I_ALIAS

Create Device Alias

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_alias_pb {
   syscb cb;
   u_char *alias_name,
      *real_name;
} i_alias_pb, *I_alias_pb;
```

Description

`I_ALIAS` creates an alternate name for a device pathlist. Processes can then reference a specific device pathlist with a shorter or more convenient name.

To delete an existing alias from the system, pass a `NULL` pointer for the real name.

Do not use a real device name as `alias_name`.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`
is the control block header.

`alias_name`
points to the alternate name.

`real_name`
points to the actual device name; it must exist. OS-9 does not validate its existence of the device.

Possible Errors

`EOS_BPNAM`

I_ATTACH

Attach New Device to System

Headers

```
#include <io.h>
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_attach_pb {
   syscb cb;
   u_char *name;
   u_int16 mode;
   Dev_list dev_tbl;
} i_attach_pb, *I_attach_pb;
```

Description

I_ATTACH causes a new I/O device to become known to the system or verifies the device is already attached.

If the descriptor is found and the device is not already attached, I_ATTACH links to its file manager and device driver and places their addresses in a new device list entry. I_ATTACH allocates and initializes static storage memory for the file manager and device driver. After initialization, the file manager's I_ATTACH entry point is called to allow for file manager specific initialization. In turn, the file manager calls the driver's initialization entry point to initialize the hardware. If the driver has already been attached, the file manager usually omits calling the driver.

I_ATTACH prepares the device for subsequent use by any process, but does not reserve the device. I_ATTACH is not required to perform routine I/O.

IOMAN attaches all devices at I_OPEN and detaches them at I_CLOSE.

Attach and Detach for devices are used together like Link and Unlink for modules. However, you can improve system performance slightly by attaching all devices at startup. This increments each device's use count and prevents the device from being reinitialized every time it is opened. If static storage for devices is allocated all at once, memory fragmentation is minimized. If a device is attached, the termination routine is not executed until the device is detached.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`name` points to the I/O device. `name` is used to search the current module directory for a device descriptor module with the same name 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, and other related information.

`mode` is the access mode used to verify subsequent read and/or write operations are permitted. It can be either `S_IREAD` or `S_IWRITE`.

`dev_tbl` is a returned value. It points to the device's device list entry.

Possible Errors

EOS_BMODE
EOS_DEVBSY
EOS_DEVOVF
EOS_MEMFUL

See Also

I_CLOSE
I_DETACH
I_OPEN

I_CHDIR

Change Working Directory

Headers

```
#include <types.h>
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_chdir_pb {
   syscb cb;
   u_char *name;
   u_int16 mode;
} i_chdir_pb, *I_chdir_pb;
```

Description

`I_CHDIR` changes a process' working directory to the directory file specified by the pathlist. The execution or data directory (or both) may be changed, depending on the specified access mode. The file specified must be a directory file, and the caller must have access permission for the specified mode.

If the access mode is read, write, or update (read and write), the current data directory is changed. If the access mode is execute, the current execution directory is changed. You can change both simultaneously.

The shell `chd` directive uses update mode. This means you must have both read and write permission to change directories from the shell. This is a recommended practice.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`name`	points to the pathlist.

`mode` specifies the access mode. The following are the valid modes:

Mode	Description
S_IREAD	Read
S_IWRITE	Write
S_IEXEC	Execute

Possible Errors

EOS_BMODE
EOS_BPNAM

I_CIOPROC

Get Pointer to I/O Process Descriptor

Headers

```
#include <io.h>
```

Parameter Block Structure

```
typedef struct i_cioproc_pb {
   syscb cb;
   process_id proc_id;
   void *buffer;
   u_int32 count;
} i_cioproc_pb, *I_cioproc_pb;
```

Description

`I_CIOPROC` copies the I/O process descriptor for the specified process into a buffer.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`proc_id`	is the process ID of the process.
`buffer`	points to the buffer in which to copy the process descriptor.
`count`	specifies the number of bytes to copy.

Possible Errors

`EOS_IPRCID`

I_CLOSE

Close Path to File/Device

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_close_pb {
   syscb cb;
   path_id path;
} i_close_pb, *I_close_pb;
```

Description

I_CLOSE terminates an I/O path.

The path number is no longer valid for OS-9 calls unless it becomes active again through an I_OPEN, I_CREATE, or I_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 internally managed buffers and descriptors are deallocated.

F_EXIT automatically closes any open paths. By convention, standard I/O paths are not closed unless it is desired to change the corresponding files/devices.

I_CLOSE does an implied I_DETACH call. If this causes the device use count to become zero, the device termination routine is executed.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

cb	is the control block header.
path	identifies the I/O path to close.

Possible Errors

EOS_BPNUM

See Also

F_EXIT
I_DETACH
I_DUP

I_CONFIG

Configure an Element of Process/System I/O

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_config_pb {
   syscb cb;
   u_int32 code;
   void *param;
} i_config_pb, *I_config_pb;
```

Description

I_CONFIG is a wildcard call used to configure elements of the I/O subsystem that may or may not be associated with an existing path. It is intended to be used to dynamically reconfigure system I/O resources at runtime. The target I/O resources may be system-wide resources or they may be process- or path-specific, depending on the nature of the configuration call being made.

The following sub-code with the associated parameter and defined function.

Code	Parameter	Function
IC_PATHSZ	param points to the number of additional paths the process wants beyond its initial 32.	Increases the number of paths the current process may have open beyond its initial 32. This call may only be used to increase the number of paths a process may have. It cannot be used to reduce the number of available paths.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`code`	identifies the target configuration code.
`*param`	points to any additional parameters required by the specified configuration function.

See Also

`F_CONFIG`

I_CREATE

Create Path to New File

Headers

```
#include <types.h>
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_create_pb {
   syscb cb;
   u_char *name;
   u_int16 mode;
   path_id path;
   u_int32 perm,
      size;
} i_create_pb, *I_create_pb;
```

Description

`I_CREATE` creates a new file. On multi-file devices, the new file name is entered in the directory structure. On non-multi-file devices, `I_CREATE` is synonymous with `I_OPEN`. Also, if the file already exists on a multi-file device, by default a path to the file will be opened and the contents truncated.

`mode` must have the write bit set if data is to be written to the file. The file is given the attributes passed in `perm`. The individual bits are defined as follows:

Mode Bits	Attribute Bits
S_IREAD = read	S_IREAD = owner read permission
S_IWRITE = write	S_IWRITE = owner write permission
S_IEXEC = execute	S_IEXEC = owner exec permission
S_ICONTIG = ensure contig	S_IGREAD = group read permission
S_IEXCL = do not recreate	S_IGWRITE = group write permission
S_IAPPEND = append to file	S_IGEXEC = group exec permission
S_ISHARE = exclusive use	S_IOREAD = public read permission
S_ISIZE = set initial size	S_IOWRITE = public write permission
	S_IOEXEC = public exec permission
	S_ISHARE = file is non-sharable

If the `S_IEXEC` (execute) bit of the access mode byte is set, the working execution directory is searched first, instead of the working data directory.

If the `S_IEXCL` mode bit is not set and the target file already exists, the file is truncated to zero length.

If the `S_ICONTIG` mode bit is set, the space for the file is allocated from a single contiguous block.

If the `S_IAPPEND` mode bit is set and the target file already exists, the file is opened and the associated file pointer points to the end of the file.

If the `S_ISHARE` mode bit is set, the opening process has exclusive access to the file.

If the `S_ISIZE` mode bit is set, it is assumed the `size` parameter contains the initial file size of the target file.

File space is allocated automatically by `I_WRITE` or explicitly by an `I_SETSTAT` call.

If the pathlist specifies a file name that already exists, an error occurs. You cannot use `I_CREATE` to make directory files (see `I_MAKDIR`).

`I_CREATE` causes an implicit `I_ATTACH` call. The device's initialization routine is executed if the device has not been attached previously.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`name` points to the pathname of the new file.

`mode` specifies the access mode. If data is to be written to the file, `mode` must have the write bit set.

`path` is a returned value. It is the path number that identifies the file in subsequent I/O service requests until the file is closed.

`perm` specifies the attributes to use for the new file.

`size`	specifies the size of the new file. If the `S_ISIZE` (initial file size) bit is set, you may pass an initial file size estimate in `size`.

Possible Errors

`EOS_BPNAM`
`EOS_PTHFUL`

See Also

`I_ATTACH`
`I_CLOSE`
`I_MAKDIR`
`I_OPEN`
`I_SETSTAT`
`I_WRITE`

I_DELETE

Delete File

Headers

```
#include <types.h>
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_delete_pb {
   syscb cb;
   u_char *name;
} i_delete_pb, *I_delete_pb;
```

Description

I_DELETE deletes the file specified by the pathlist. You must have non-sharable write access to the file (the file may not already be open) or an error results. Attempts to delete non-multi-file devices result in an error.

The access mode is ignored if a full pathlist is specified (a full pathlist begins with a slash (/)).

Parameters

cb	is the control block header.
name	points to the file to delete.
mode	specifies the access mode. mode may be S_IREAD, S_IWRITE, or S_IEXEC. The access mode specifies the data or execution directory (but not both) in the absence of a full pathlist. If the access mode is read, write, or update (read and write), the current data directory is assumed. If the execute bit is set, the current execution directory is assumed.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Possible Errors

EOS_BPNAM

See Also

I_ATTACH
I_CREATE
I_DETACH
I_OPEN

I_DETACH

Remove Device from System

Headers

```
#include <io.h>
```

Parameter Block Structure

```
typedef struct i_detach_pb {
   syscb cb;
   Dev_list dev_tbl;
} i_detach_pb, *I_detach_pb;
```

Description

I_DETACH removes a device from the system device list if the device is not in use by any other process.

If this is the last use of the device, the file manager's I_DETACH routine is called, and in turn, the device driver's termination routine is called and any permanent storage assigned to the file manager and 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.

I_DETACH must be used to detach devices attached with I_ATTACH. Both of these attach and detach requests are used mainly by IOMAN 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 while the system is up. An infrequently used device can be attached and then detached to free system resources when no longer needed.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt
Threads:	Safe

Parameters

`cb`	is the control block header.
`dev_tbl`	points to the address of the device list entry.

See Also

`I_ATTACH`
`I_CLOSE`

I_DUP

Duplicate Path

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_dup_pb {
   syscb cb;
   path_id dup_path,
      *new_path;
} i_dup_pb, *I_dup_pb;
```

Description

I_DUP duplicates a path. The operation of I_DUP depends on the state from which it is called.

When called from a user-state process and given an existing path number, 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 perform an I_CLOSE on path 0 and an 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.

When called from a system-state process, I_DUP returns the next available system path number.

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.

I_DUP 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 performing I/O on the same path concurrently. On RBF files, this can produce unpredictable results.

Attributes

Operating System:	OS-9
State:	User, System, and Interrupt

Threads:	Safe

Parameters

`cb`	is the control block header.
`dup_path`	is the path number of the path to duplicate.
`new_path`	is the new number for the same path.

Possible Errors

`EOS_BPNUM`
`EOS_PTHFUL`

See Also

`I_CLOSE`

I_GETDL

Get System I/O Device List Head Pointer

Headers

```
#include<io.h>
```

Parameter Block Structure

```
typedef struct i_getdl_pb{
   syscb cb;
   Dev_list dev_list_ptr;
} i_getdl_pb, *I_getdl_pb;
```

Description

`I_GETDL` returns a pointer to the first entry in the system's I/O device list.

Attributes

Operating System:	OS-9
State:	User, System, I/O, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`dev_list_ptr` is a returned value. It points to the first entry in the device list.

Never access this pointer directly in user state. You should use `F_CPYMEM` to get a copy of the device list entry. This system call is used by the `devs` utility to determine the presence of all of the active devices in the system.

See Also

`F_CPYMEM`

I_GETPD

Find Path Descriptor

Headers

```
#include <types.h>
#include <io.h>
```

Parameter Block Structure

```
typedef struct i_getpd_pb {
   syscb cb;
   path_id path;
   Pd_com path_desc;
} i_getpd_pb, *I_getpd_pb;
```

Description

`I_GETPD` converts a path number to the absolute address of its path descriptor data structure.

Attributes

Operating System:	OS-9
State:	System, I/O, and Interrupt
Threads:	Safe

Parameters

`cb` is the control block header.

`path` specifies the path number.

`path-id` is a returned value. It points to the path descriptor.

I_GETSTAT

Get File/Device Status

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct i_getstat_pb {
   syscb cb;
   path_id path;
   u_int16 gs_code;
   void *param_blk;
} i_getstat_pb, *I_getstat_pb;
```

Description

`I_GETSTAT` 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 often used with `I_SETSTAT`, which sets the device operating parameters.

The mnemonics for the status codes are found in the header file `funcs.h`. Codes 0 - 127 are reserved for Microware’s use. You may define the remaining codes and their parameter passing conventions. The status codes that are currently defined and the functions they perform are described in the functions with an `SS_` prefix.

Supported getstats include:

Getstat	Description
`I_GETSTAT, SS_COPYPD`	Copy Contents of Path Descriptor (All)
`I_GETSTAT, SS_CSTATS`	Get Cache Status Information (RBF)
`I_GETSTAT, SS_DEVNAME`	Return Device Name (All)
`I_GETSTAT, SS_DEVOPT`	Read Device Path Options
`I_GETSTAT, SS_DEVTYPE`	Return Device Type (All)
`I_GETSTAT, SS_DSIZE`	Get Size of SCSI Devices (RBF)
`I_GETSTAT, SS_EDT`	Get I/O Interface Edition Number (All)

Getstat	Description
I_GETSTAT, SS_EOF	Test for End of File (All)
I_GETSTAT, SS_FD	Read File Descriptor Sector (RBF, PIPE)
I_GETSTAT, SS_FdAddr	Get File Descriptor Block Address for Open File (RBF, PCF)
I_GETSTAT, SS_FDINFO	Get Specified File Descriptor Sector (RBF, Pipe)
I_GETSTAT, SS_LUOPT	Read Logical Unit Options (All)
I_GETSTAT, SS_PARITY	Calculate Parity of File Descriptor (RBF)
I_GETSTAT, SS_PATHOPT	Read Path Descriptor Option Section (All)
I_GETSTAT, SS_POS	Get Current File Position (RBF)
I_GETSTAT, SS_READY	Test for Data Ready (RBF, SCF, PIPE)
I_GETSTAT, SS_SIZE	Set File Size (RBF, PIPE, PCF)

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb` is the control block header.

`path` is the path number.

`gs_code` is the get status code.

`param_blk` points to the parameter block corresponding to the function being performed. If the get status function does not require a parameter block, `param_blk` should be null.

Possible Errors

EOS_UNKSVC

See Also

I_SETSTAT

I_GETSTAT, SS_COPYPD

Copy Contents of Path Descriptor (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_cpypd_pb {
   u_int32 size;
   void *path_desc;
} gs_cpypd_pb, *Gs_cpypd_pb;
```

Description

`SS_COPYPD` copies the contents of the specified path's path descriptor to the path descriptor buffer.

Attributes

Operating System:	OS-9
State:	User, System, I/O, and Interrupt
Threads:	Safe

Parameters

`size` is the number of bytes to copy from the path descriptor. If the `size` value is greater than the size of the target path descriptor, `size` is updated with the actual size of the path descriptor.

`path_desc` points to the buffer for the path descriptor data.

Possible Errors

`EOS_BPNUM`

I_GETSTAT, SS_CSTATS

Get Cache Status Information (RBF)

Headers

```
#include <rbf.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_cstats_pb {
   Cachestats cache_inf;
} gs_cstats_pb, *Gs_cstats_pb;
```

Description

`SS_CSTATS` returns a copy of the current `cachestats` structure.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cache_inf`	points to a structure containing information about RBF caching.

Possible Errors

`EOS_BPNUM`

I_GETSTAT, SS_DEVNAME

Return Device Name (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_devname_pb {
    u_char *namebuf;
} gs_devname_pb, *Gs_devname_pb;
```

Description

SS_DEVNAME returns the name of the device associated with the specified path.

Attributes

Operating System:	OS-9
State:	User, System, I/O, and Interrupt
Threads:	Safe

Parameters

namebuf	points to the buffer containing the device name.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_DEVOPT

Read Device Path Options

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_dopt_pb {
  u_int32 dopt_size;
  void *user_dopts;
} gs_dopt_pb,  *Gs_dopt_pb;
```

Description

SS_DEVOPT gets the initial (default) device path options. These options are used for initializing new paths to the device.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

dopt_size — is a returned value. It is the size of the option area.

user_dopts — points to the list of device path options buffer.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_DEVTYPE

Return Device Type (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_devtype_pb {
  u_int16 type;
  u_int16 class;
} gs_devtype_pb, *Gs_devtype_pb;
```

Description

SS_DEVTYPE returns the type and class of the device associated with the specified path number.

The values for the device type and device class are defined in the io.h header file.

Attributes

Operating System:	OS-9
State:	User, System, I/O, and Interrupt
Threads:	Safe

Parameters

type
: is a returned value. It is the device type.

class
: is a returned value. It is the device class.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_DISKFREE

Return Information About RBF Disk Free Space

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_diskfree_pb {
  u_int32     bavail;
  u_int32     bsize;
  u_int32     blocks;
  u_int32     bcontig;
} gs_diskfree_pb, *Gs_diskfree_pb;
```

Description

`SS_DISKFREE` returns information about RBF disk free space.

Attributes

Operating System:	OS-9
State:	User and System
Threads:	Safe

Parameters

`bavail`
set to total number of free blocks on the disk.

`bsize`
set to size of blocks used on the disk (256, 512, etc.).

`blocks`
set to total number of blocks on the disk.

`bcontig`
set to number of blocks in the largest contiguous area.

Possible Errors

`EOS_READ`

I_GETSTAT, SS_DSIZE

Get Size of SCSI Devices (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_dsize_pb {
  u_int32 totblocks,
     blocksize;
} gs_dsize_pb, *Gs_dsize_pb;
```

Description

SS_DSIZE gets information about the size of a SCSI disk drive.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`totblocks` is a returned value. It is the total number of blocks on the device.

`blocksize` is a returned value. It is the size of a disk block in bytes.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_EDT

Get I/O Interface Edition Number (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_edt_pb {
  u_int32 edition;
} gs_edt_pb, *Gs_edt_pb;
```

Description

SS_EDT returns the I/O interface edition number of the driver. It validates the compatibility of drivers and file managers.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

edition	is the driver I/O interface edition number.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_EOF

Test for End of File (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_eof_pb {
   u_int32 eof;
} gs_eof_pb, *Gs_eof_pb;
```

Description

SS_EOF returns the EOS_EOF error if the current position of the file pointer associated with the specified path is at the end-of-file. SCF never returns EOS_EOF.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

eof

is the end-of-file status of the specified path. A value of 1 indicates end of file.

Possible Errors

EOS_BPNUM
EOS_EOF

I_GETSTAT, SS_FD

Read File Descriptor Sector (RBF, PIPE)

Headers

```
#include <types.h>
#include <rbf.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_fd_pb {
  u_int32 info_size;
  Fd_stats fd_info;
} gs_fd_pb, *Gs_fd_pb;
```

Description

`SS_FD` returns a copy of the file descriptor sector for the file associated with the specified path.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`infosize` is the number of bytes of the file descriptor to copy.

`fdinfo` points to the buffer for the file descriptor sector.

Possible Errors

`EOS_BPNUM`

I_GETSTAT, SS_FdAddr

Get File Descriptor Block Address for Open File (RBF, PCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_fdaddr_pb {
  u_int32 fd_blkaddr;
} gs_fdaddr_pb, *Gs_fdaddr_pb;
```

Description

`SS_FdAddr` returns the file descriptor block address associated with the specified path number.

Only super users can make this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`fd_blkaddr`	is the block address of the file descriptor.

Possible Errors

`EOS_BPNUM`
`EOS_PERMIT`

I_GETSTAT, SS_FDINFO

Get Specified File Descriptor Sector (RBF, PIPE)

Headers

```
#include <rbf.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_fdinf_pb {
  u_int32 info_size,
    fd_blk_num;
  Fd_stats fd_info;
} gs_fdinf_pb, *Gs_fdinf_pb;
```

Description

SS_FDINFO returns a copy of the specified file descriptor sector for the file associated with the specified path.

Typically, SS_FDINFO is used to rapidly scan a directory on a device. You do not need to specify the path number of the file for which you want the file descriptor. However, the path number must be an open path on the same device as the file. The path number typically represents a path to the directory you are currently scanning.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

info_size	specifies the number of bytes of the file descriptor block to copy.
fd_blk_num	specifies the file descriptor sector number to get.
fd_info	points to the buffer for the file descriptor block.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_LUOPT

Read Logical Unit Options (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_luopt_pb {
   u_int32 luopt_size;
   void *user_luopts;
} gs_luopt_pb, *Gs_luopt_pb;
```

Description

`SS_LUOPT` copies the contents of the logical unit options for a path into the options buffer.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`luopt_size`
the size of the options section to copy. `luopt_size` may not be less than the size of the file manager's logical unit option section.

`user_luopts`
points to the options buffer.

Possible Errors

```
EOS_BPNUM
EOS_BUF2SMALL
```

I_GETSTAT, SS_PARITY

Calculate Parity of File Descriptor (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_parity_pb {
  Fd_status fd;
  u_int16 parity;
} gs_parity_pb, *Gs_parity_pb;
```

Description

SS_PARITY calculates a 32 bit vertical parity for file descriptor structures. This call is used by utilities creating disk images (format disks) and utilities checking the integrity of disks.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

fd	points to the file descriptor block.
parity	is the resulting parity.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_PATHOPT

Read Path Descriptor Option Section (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_popt_pb {
   u_int32 popt_size;
   void *user_popts;
} gs_popt_pb, *Gs_popt_pb;
```

Description

SS_PATHOPT copies the option section of the path descriptor into the variable-sized area options buffer. You must include rbf.h, sbf.h, and/or scf.h for the corresponding file managers and to declare popt_size according to the size of the rbf_opts, sbf_opts, or scf_opts. SS_PATHOPT is typically used to determine the current settings for functions such as echo and auto line feed.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

popt_size	is the size of the path options section to copy.
user_opts	points to the options buffer.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_POS

Get Current File Position (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_pos_pb {
  u_int32 filepos;
} gs_pos_pb, *Gs_pos_pb;
```

Description

SS_POS returns the current position of the file pointer associated with the specified path.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

filepos	is the file position in byte-size units.

Possible Errors

EOS_BPNUM

I_GETSTAT, SS_READY

Test for Data Ready (RBF,SCF, PIPE)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_ready_pb {
   u_int32 incount;
} gs_ready_pb, *Gs_ready_pb;
```

Description

SS_READY checks for data available to be read on the specified path. The number of characters available to be read is returned in the incount parameter. RBF devices do not return the EOS_NRDY error. SS_READY returns the number of bytes left in the file and SUCCESS.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

incount	is the number of characters available to be read.

Possible Errors

EOS_BPNUM
EOS_NRDY

I_GETSTAT, SS_SIZE

Set File Size (RBF, PIPE, PCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct gs_size_pb {
    u_int32 filesize;
}  gs_size_pb, *Gs_size_pb;
```

Description

SS_SIZE gets the size of the file associated with the open path to the specified filesize.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

filesize	is the new size of the file in bytes.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_GIOPROC

Get Pointer to I/O Process Descriptor

Headers

```
#include <io.h>
```

Parameter Block Structure

```
typedef struct i_cioproc_pb {
   syscb cb;
   process_id proc_id;
   Io_proc proc_desc;
} i_cioproc_pb, *I_cioproc_pb;
```

Description

I_GIOPROC returns a pointer to the I/O process descriptor for the process specified.

Attributes

Operating System:	OS-9
State:	System and I/O
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` specifies the process ID of the process.

`proc_desc` is a returned value. It points to the I/O process descriptor.

Possible Errors

EOS_IPRCIDT

I_IODEL

Check for Use of I/O Module

Headers

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

Parameter Block Structure

```
typedef struct i_iodel_pb {
   syscb cb;
   Mh_com mod_head;
} i_iodel_pb, *I_iodel_pb;
```

Description

`I_IODEL` is executed whenever the kernel unlinks a file manager, device driver, or device descriptor module. It is used to determine if the I/O system is still using the module.

Attributes

Operating System:	OS-9
State:	System and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`mod_head`	points to the module header.

Possible Errors

`EOS_MODBSY`

I_IOEXIT

Terminate I/O for Exiting Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_ioexit_pb {
   syscb cb;
   process_id proc_id;
   u_int32 path_cnt;
} i_ioexit_pb, *I_ioexit_pb;
```

Description

`I_IOEXIT` is executed whenever the kernel terminates or chains to a process.

Attributes

Operating System:	OS-9
State:	System and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`proc_id`	specifies the process ID.

`path_cnt` specifies the number of I/O paths.

If the most significant bit of `path_cnt` is reset, the process’ default data and execution directory paths and all other open paths in the path translation table are closed. The I/O process descriptor is also deallocated.

If the most significant bit of `path_cnt` is set, the remaining bits specify the number of paths to leave open. The default directory paths are not closed, and the I/O process descriptor is not deallocated.

Possible Errors

`EOS_IPRCID`

I_IOFORK

Set Up I/O for New Process

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_iofork_pb {
   syscb cb;
   process_id par_proc_id,
      new_proc_id;
   u_int32 path_cnt;
}  i_iofork_pb, *I_iofork_pb;
```

Description

I_IOFORK is executed whenever the kernel creates a new process. I_IOFORK creates an I/O process descriptor for the new process. IOMAN uses I/O process descriptors to maintain information about a process' I/O. Each I/O process descriptor contains the user-to-system path number translation table and path numbers for the process' default data and execution directories.

Attributes

Operating System:	OS-9
State:	System and I/O
Threads:	Safe

Parameters

cb	is the control block header.
par_proc_id	is the parent's process ID.
new_proc_id	is the process ID of the new process.
path_cnt	is the number of I/O paths the child is to inherit from its parent.

Possible Errors

EOS_NORAM

I_MAKDIR

Make New Directory

Headers

```
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_makdir_pb {
   syscb cb;
   u_char *name;
   u_int16 mode;
   u_int32 perm,
      size;
} i_makdir_pb, *I_makdir_pb;
```

Description

I_MAKDIR creates and initializes a new directory as specified by the pathlist. I_MAKDIR is the only way to create a new directory file. The new directory file contains only entries for itself (.) and its parent directory (..). 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. If the pathlist begins with a slash, it is used as the pathlist.

The caller becomes the owner of the directory. I_MAKDIR does not return a path number because directory files are not opened by this request. You should use I_OPEN to open a directory.

The new directory automatically has its *directory* bit set in the access permission attributes. The remaining attributes are specified by the bytes passed in the mode and perm parameters. The individual bits for these parameters are defined as follows (if the bit is set, access is permitted):

Mode Bits	Attribute Bits
S_IREAD = read	S_IREAD = owner read permission
S_IWRITE = write	S_IWRITE = owner write permission
S_IEXEC = execute	S_IEXEC = owner exec permission
S_ITRUNC = truncate on open	S_IGREAD = group read permission
S_ICONTIG = ensure contig	S_IGWRITE = group write permission
S_IEXCL = do not recreate	S_IGEXEC = group exec permission

Mode Bits	Attribute Bits
S_IAPPEND = append to file	S_IOREAD = public read permission
S_ISHARE = exclusive use	S_IOWRITE = public write permission
S_ISIZE = set initial size	S_IOEXEC = public exec permission
	S_ISHARE = file is non-sharable

If the `S_IEXEC` (execute) bit of the access mode byte is set, the working execution directory is searched first instead of the working data directory.

If the `S_IEXCL` mode bit is not set and the target file already exists, the file is truncated to zero length.

If the `S_ICONTIG` mode bit is set, the space for the file is allocated from a single contiguous block.

If the `S_ITRUNC` mode bit is set and the target file already exists, the file is truncated to zero length.

If the `S_IAPPEND` mode bit is set and the target file already exists, the file is opened and the associated file pointer points to the end of the file.

If the `S_ISHARE` mode bit is set, the opening process has exclusive access to the file.

If the `S_ISIZE` mode bit is set, it is assumed the `size` parameter contains the initial file size of the target file.

Attributes

Operating System:	OS-9
State:	User and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`name`	points to the pathlist.
`mode`	specifies the access mode.
`perm`	specifies the access permissions.
`size`	is optional; it specifies the initial allocation size.

Possible Errors

EOS_BPNAM
EOS_CEF
EOS_FULL

See Also

I_OPEN

I_OPEN

Open Path to File or Device

Headers

```
#include <types.h>
#include <modes.h>
```

Parameter Block Structure

```
typedef struct i_open_pb {
   syscb cb;
   u_char *name;
   u_int16 mode;
   path_id path;
} i_open_pb, *I_open_pb;
```

Description

I_OPEN opens a path to an existing file or device as specified by the pathlist. I_OPEN returns a path number used in subsequent service requests to identify the path. If the file does not exist, an error is returned.

A non-directory file may be opened with no bits set. This allows you to use the I_GETSTAT system requests to examine characteristics such as attributes and size, but does not permit any actual I/O on the path.

For RBF devices, use Read mode instead of Update if the file is not going to be modified. This inhibits record locking and can dramatically improve system performance if more than one user is accessing the file. The access mode must conform to the access permissions associated with the file or device (see I_CREATE).

Mode	Description
S_IREAD	Read
S_IWRITE	Write
S_IEXEC	Execute
S_ISHARE	Open file for non-sharable use
S_IFDIR	Open directory file

Refer to `modes.h` for more information about the modes available for `I_OPEN`.

If the execution bit mode is set, OS-9 searches for the file in the working execution directory, unless the pathlist begins with a slash. If the pathlist begins with a slash, it uses the entire pathlist and opens the file or device with the execute bit set.

`I_OPEN` searches only for executables in the execution directory if the `FAM_EXEC` access mode is used. The execution directory is designed for the location of executable modules, not data modules. The access determination is done by IOMAN based on the file permissions. To override this behavior, add `S_IEXEC` to the file creation permissions.

If the single user bit is set, the file is opened for non-sharable access even if the file is sharable.

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.

Directory files may be opened only if the directory bit (`S_IFDIR`) is set in the access mode.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb` is the control block header.

`name` points to the path name of the existing file or device.

`mode` specifies which subsequent read and/or write operations are permitted as follows (if the bit is set, access is permitted).

`path` is the resulting path number.

Possible Errors

EOS_BMODE
EOS_BPNAM
EOS_FNA
EOS_PNNF
EOS_PTHFUL
EOS_SHARE

See Also

I_ATTACH
I_CLOSE
I_CREATE
I_GETSTAT

I_RDALST

Copy System Alias List

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_rdalst_pb {
    syscb cb;
    u_char *buffer;
    u_int32 count;
} i_rdalst_pb, *I_rdalst_pb;
```

Description

I_RDALST copies the system alias list to the caller's buffer. At most, count bytes are copied to the buffer. Each alias entry is null terminated.

The I_RDALST system call is used by the alias utility to display the list of aliases currently active in the system.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

cb	is the control block header.
buffer	points to the buffer into which to copy the alias list.
count	is the total number of bytes to copy. count is updated with the total number of bytes copied.

Possible Errors

EOS_BPADDR

See Also

I_ALIAS

I_READ

Read Data from File or Device

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_read_pb {
   syscb cb;
   path_id path;
   u_char *buffer;
   u_int32 count;
} i_read_pb, *I_read_pb;
```

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 not enough data is 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.

The keyboard X-ON/X-OFF characters may be filtered out of the input data on SCF-type devices unless the corresponding entries in the path descriptor have been set to zero. You may want to modify the device descriptor so these path descriptor values are initialized to zero when the path is opened. SCF devices usually terminate the read request when a carriage return is reached.

For RBF devices, if the file is open for update, the record read is locked out. For more information, refer to the Record Locking section in **Chapter 5: OS-9 File System**.

The number of bytes requested are read unless the end-of-file is reached, an end-of-record occurs (SCF only), the read times out (SCF only), or an error condition occurs.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb` is the control block header.

`path` specifies the path number.

`buffer` points to the data buffer.

`count` is the number of bytes to read. Upon completion, `count` is updated with the number of bytes actually read.

Possible Errors

```
EOS_BMODE
EOS_BPNUM
EOS_EOF
EOS_READ
```

See Also

`I_READLN`

I_READLN

Read Text Line with Editing

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_readln_pb {
   syscb cb;
   path_id path;
   u_char *buffer;
   u_int32 count;
} i_readln_pb, *I_readln_pb;
```

Description

I_READLN reads the specified number of bytes from the input file or device until an end-of-line character is encountered. On SCF-type devices, I_READLN also causes line editing such as backspacing, line delete, echo, and automatic line feed to occur. Some SCF devices 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, an I_READLN of exactly one byte accepts only a carriage return to return without error and beeps when other keys are pressed. An I_READLN to SCF returns the number of bytes requested unless the read times out or an error occurs.

After all data in a file has been read, the next I_READLN service request returns an end of file error.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`path`	specifies the path number.
`buffer`	points to the data buffer.
`count`	is the number of bytes to read. Upon completion, `count` is updated with the number of bytes actually read.

Possible Errors

```
EOS_BMODE
EOS_BPNUM
EOS_EOF
EOS_READ
```

See Also

`I_READ`

I_SEEK

Reposition Logical File Pointer

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_seek_pb {
   syscb cb;
   path_id path;
   u_int32 offset;
} i_seek_pb, *I_seek_pb;
```

Description

I_SEEK repositions the path’s file pointer. The file pointer is the 32-bit address of the next byte in the file to be read or written. I_SEEK usually does not initiate physical positioning of the media. You can perform a seek to any value, even if the file is not large enough. Subsequent write requests automatically expand the file to the required size, if possible. Read requests return an end-of-file condition.

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.

On RBF devices, seeking to a new disk sector rewrites the internal sector buffer to disk if it has been modified. I_SEEK does not change the state of record locks. Beware of seeking to a negative position. RBF interprets negatives as large positive numbers.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`path`	specifies the path number.
`position`	specifies the new position.

Possible Errors

EOS_BPNUM

See Also

I_READ
I_WRITE

I_SETSTAT

Set File/Device Status

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct i_seek_pb {
   syscb cb;
   path_id path;
   u_int16 ss_code;
   void *param_blk;
} i_seek_pb, *I_setstat_pb;
```

Description

I_SETSTAT is a wildcard call used to handle individual device parameters that are not uniform on all devices or are highly hardware dependent.

Typically, set status calls are used to set a terminal's parameters for functions such as backspace character, delete character, echo on/off, null padding, and paging. I_SETSTAT is commonly used with I_GETSTAT which reads the device's operating parameters. The mnemonics for the status codes are found in the header file funcs.h. Codes 0-127 are reserved for Microware's use. Users may define the remaining codes and their parameter passing conventions.

Supported setstats include:

Setstat	Description
I_SETSTAT, SS_ATTR	Set File Attributes (RBF, Pipe, PCF)
I_SETSTAT, SS_BREAK	Break Serial Connection (SCF)
I_SETSTAT, SS_CACHE	Enable/Disable RBF Caching (RBF)
I_SETSTAT, SS_DCOFF	Send Signal When Data Carrier Detect Line Goes False (SCF)
I_SETSTAT, SS_DCON	Send Signal When Data Carrier Detect Line Goes True (SCF)
I_SETSTAT, SS_DEVOPT	Set Device Path Options (Pipe, SBF, SCF)
I_SETSTAT, SS_DSRTS	Disable RTS Line

Setstat	Description
`I_SETSTAT, SS_ENRTS`	Enable RTS Line
`I_SETSTAT, SS_ERASE`	Erase Tape (SBF)
`I_SETSTAT, SS_FD`	Write File Descriptor Sector (RBF, PCF, PIPE)
`I_SETSTAT, SS_FILLBUFF`	Fill Path Buffer With Data (SCF)
`I_SETSTAT, SS_FLUSHMAP`	Flush Cached Bit Map Information (RBF)
`I_SETSTAT, SS_HDLINK`	Make Hard Link to Existing File (RBF)
`I_SETSTAT, SS_LOCK`	Lock Out Record (RBF)
`I_SETSTAT, SS_LUOPT`	Write Logical Unit Options (All)
`I_SETSTAT, SS_PATHOPT`	Write Option Section of Path Descriptor (All)
`I_SETSTAT, SS_RELEASE`	Release Device (SCF, PIPE)
`I_SETSTAT, SS_RENAME`	Rename File (RBF, PIPE, SCF)
`I_SETSTAT, SS_RESET`	Restore Head to Track Zero (RBF, SBF, PCF)
`I_SETSTAT, SS_RETEN`	Re-tension Pass on Tape Device (SBF)
`I_SETSTAT, SS_RFM`	Skip Tape Marks (SBF)
`I_SETSTAT, SS_SENDSIG`	Send Signal on Data Ready (SCF, PIPE)
`I_SETSTAT, SS_SIZE`	Set File Size (RBF, PIPE, PCF)
`I_SETSTAT, SS_SKIP`	Skip Blocks (SBF)
`I_SETSTAT, SS_SKIPEND`	Skip to End of Tape (SBF)
`I_SETSTAT, SS_TICKS`	Wait Specified Number of Ticks for Record Release (RBF)
`I_SETSTAT, SS_WFM`	Write Tape Marks (SBF)
`I_SETSTAT, SS_WTRACK`	Write (Format) Track (RBF)

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`path`	is the path number.

`ss_code` is the set status code.

`param_blk` points to the parameter block corresponding to the function being performed. If the set status function does not require a parameter block, `param_blk` should be `NULL`.

Possible Errors

`EOS_UNKSVC`

See Also

`I_GETSTAT`

I_SETSTAT, SS_ATTR

Set File Attributes (RBF, PIPE, PCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_attr_pb {
  u_int32 attr;
}  ss_attr_pb, *Ss_attr_pb;
```

Description

SS_ATTR changes a file's attributes to the new value, if possible. You cannot set the directory bit of a non-directory file or clear the directory bit of a non-empty directory.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

attr	specifies the file attributes to change.

Possible Errors

EOS_BPNUM

See Also

I_GETSTAT
I_SETSTAT

I_SETSTAT, SS_BREAK

Break Serial Connection (SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description:

SS_BREAK breaks a serial connection.

The driver is responsible for implementing this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_SETSTAT, SS_CACHE

Enable/Disable RBF Caching (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_cache_pb {
  u_int32 enblflag,
     drvscize;
}  ss_cache_pb, *Ss_cache_pb;
```

Description

SS_CACHE enables and disables RBF caching on an RBF device.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

enblflag is the cache enable/disable flag.

- If enblflag is zero, caching is disabled.
- If enblflag is non-zero, caching is enabled.

drvcsize is the memory size for the cache.

Possible Errors

```
EOS_CEF
EOS_PERMIT
```

See Also

I_SETSTAT

I_SETSTAT, SS_DCOFF

Send Signal When Data Carrier Detect Line Goes False (SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_dcoff_pb {
   signal_code signal;
}  ss_dcoff_pb, *Ss_dcoff_pb;
```

Description

When a modem has finished receiving data from a carrier, the Data Carrier Detect line becomes false. SS_DCOFF sends a signal code when this happens. I_SETSTAT, SS_DCON sends a signal when the line becomes true.

The driver is responsible for implementing this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`signal`	is the signal code to send.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT, SS_DCON
I_SETSTAT, SS_RELEASE

I_SETSTAT, SS_DCON

Send Signal When Data Carrier Detect Line Goes True (SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_dcon_pb {
   signal_code signal;
}  ss_dcon_pb, *Ss_dcon_pb;
```

Description

When a modem receives a carrier, the Data Carrier Detect line becomes true. SS_DCON sends a signal code when this happens. I_SETSTAT, SS_DCOFF sends a signal when the line becomes false.

The driver is responsible for implementing this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

signal	is the signal code to send.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT, SS_DCOFF
I_SETSTAT, SS_RELEASE

I_SETSTAT, SS_DEVOPT

Set Device Path Options (PIPE, SBF, SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_dopt_pb {
   u_int dopt_size;
   void *user_dopts;
}  ss_dopt_pb, *Ss_dopt_pb;
```

Description

SS_DOPT sets the initial (default) device path options. These options initialize new paths to the device.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

dopt_size	specifies the size of the options area to copy.
user_dopts	points to the default options for the device.

Possible Errors

EOS_BPNUM

See Also

I_GETSTAT
I_SETSTAT

I_SETSTAT, SS_DSRTS

Disable RTS Line

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to set the status parameter block.

Description

`SS_DSRTS` disables the RTS line.

The driver is responsible for implementing this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT, SS_ENRTS`

I_SETSTAT, SS_ENRTS

Enable RTS Line

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_dcoff_pb {
  signal_code signal;
}  ss_dcoff_pb, *Ss_dcoff_pb;
```

Description

SS_ENRTS asserts the RTS line.

The driver is responsible for implementing this call.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

signal	is the signal code to send.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT, SS_DSRTS

I_SETSTAT, SS_ERASE

Erase Tape (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_erase_pb {
   u_int32 blks;
}  ss_erase_pb, *Ss_erase_pb;
```

Description

SS_ERASE erases a portion of the tape. The amount of tape erased depends on the hardware capabilities.

This is dependent on both the hardware and the driver.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

blks	specifies the number of blocks to erase.

- If blks is -1, SBF erases until the end-of-tape is reached.
- If blks is positive, SBF erases the amount of tape equivalent to that number of blocks.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_SETSTAT, SS_FD

Write File Descriptor Sector (RBF, PCF, PIPE)

Headers

```
#include <rbf.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_fd_pb {
   Fd_stats fd_info;
}  ss_fd_pb, *Ss_fd_pb;
```

Description

`SS_FD` changes the file descriptor sector data. The path must be open for write.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`fd_info`	points to the file descriptor's buffer.

You can only change `fd_group`, `fd_owner`, and the time stamps `fd_atime`, `fd_mtime`, and `fd_utime`. These are the only fields written back to the disk. These fields are defined in the `fd_stats` structure in `rbf.h`. Only the super user can change the file's owner ID.

Possible Errors

`EOS_BPNUM`

See Also

`I_GETSTAT`
`I_SETSTAT`

I_SETSTAT, SS_FILLBUFF

Fill Path Buffer With Data (SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_fillbuff_pb {
   u_int32 size;
   u_char *user_buff;
}  ss_fillbuff_pb, *Ss_fillbuff_pb;
```

Description

`SS_FILLBUFF` fills the input path buffer with the data in `buffer`.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`size` specifies the size of the buffer (amount of data to copy).

`user_buff` points to the data buffer.

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT`

I_SETSTAT, SS_FLUSHMAP

Flush Cached Bit Map Information (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description

`SS_FLUSHMAP` flushes the cached bit map information for an RBF device. This normally would only be performed after the bit map on the disk is changed by a utility such as `format`.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT`

I_SETSTAT, SS_HDLINK

Make Hard Link to Existing File (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_link_pb {
    u_char *link_path;
}  ss_link_pb, *Ss_link_pb;
```

Description

SS_HDLINK creates a new directory entry specified by link_path. This directory entry points to the file descriptor block of the open file specified by path in the I_SETSTAT parameter block. SS_HDLINK updates the pathlist pointer.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

link_path
points to the new name for the directory entry.

Possible Errors

EOS_BPNUM
EOS_CEF
EOS_PNNF

See Also

I_SETSTAT

I_SETSTAT, SS_LOCK

Lock Out Record (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_lock_pb {
   u_int32 size;
}  ss_lock_pb, *Ss_lock_pb;
```

Description

SS_LOCK locks out a section of the file from the current file pointer position up to the specified number of bytes.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`size`
: is the size of the section to lockout. If `size` is zero, all locks are removed (record lock, EOF lock, and file lock). If $ffffffff bytes are requested, the entire file is locked out regardless of the file pointer's location. This is a special type of file lock that remains in effect until released by an SS_LOCK with `size` set to zero, a read or write of zero bytes, or the file is closed.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_SETSTAT, SS_LUOPT

Write Logical Unit Options (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_luopt_pb {
   u_int32 luopt_size;
   void *user_luopts;
}  ss_luopt_pb, *Ss_luopt_pb;
```

Description

SS_LUOPT writes the logical unit options for a path to a buffer.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

luopt_size	specifies the buffer size of the logical unit options area.
user_luopts	points to the logical unit options.

Possible Errors

EOS_BPNUM
EOS_BUF2SMALL

See Also

I_GETSTAT
I_SETSTAT

I_SETSTAT, SS_PATHOPT

Write Option Section of Path Descriptor (ALL)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_popt_pb {
   u_int popt_size;
   void *user_popts;
}  ss_popt_pb, *Ss_popt_pb;
```

Description

SS_PATHOPT writes the option section of the path descriptor from the status packet pointed to by user_opts. Typically, SS_PATHOPT sets the device operating parameters (such as echo and auto line feed). This call is handled by the file managers, and only copies values appropriate for user programs to change.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

popt_size	specifies the buffer size.
user_popts	points to the options buffer.

Possible Errors

EOS_BPNUM
EOS_BUF2SMALL

See Also

I_GETSTAT
I_SETSTAT

I_SETSTAT, SS_RELEASE

Release Device (SCF, PIPE)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description

`SS_RELEASE` releases the device from any `SS_SENDSIG`, `SS_DCON`, or `SS_DCOFF` request made by the calling process.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT, SS_DCOFF`
`I_SETSTAT, SS_DCON`
`I_SETSTAT, SS_SENDSIG`

I_SETSTAT, SS_RENAME

Rename File (RBF, PIPE, SCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_rename_pb {
   char *newname;
}  ss_rename_pb, *Ss_rename_pb;
```

Description

SS_RENAME changes the file name of the directory entry associated with the open path. You cannot change a file's name to that of a file already existing in a directory.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

newname	points to the file's new name.

Possible Errors

EOS_CEF

See Also

I_SETSTAT

I_SETSTAT, SS_RESET

Restore Head to Track Zero (RBF, SBF, PCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description

For RBF and PCF, `SS_RESET` directs the disk head to track zero. It is used for formatting and error recovery. For SBF, `SS_RESET` rewinds the tape.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT`

I_SETSTAT, SS_RETEN

Re-tension Pass on Tape Drive (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description

`SS_RETEN` performs a re-tension pass on the tape drive.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

```
EOS_BPNUM
EOS_NOTRDY
```

See Also

`I_SETSTAT`

I_SETSTAT, SS_RFM

Skip Tape Marks (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_rfm_pb {
   int32 cnt;
}  ss_rfm_pb, *Ss_rfm_pb;
```

Description

SS_RFM skips the number of tape marks specified.

Parameters

cnt	specifies the number of tape marks to skip. If cnt is negative, the tape is rewound the specified number of marks.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

EOS_BPNUM
EOS_NOTRDY

See Also

I_SETSTAT

I_SETSTAT, SS_SENDSIG

Send Signal on Data Ready (SCF, PIPE)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_sendsig_pb {
   signal_code signal;
}  ss_sendsig_pb, *Ss_sendsig_pb;
```

Description

SS_SENDSIG sets up a signal to be sent to a process when an interactive device or pipe has data ready. SS_SENDSIG 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.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

signal	is the signal to send.

Possible Errors

```
EOS_BMODE
EOS_BPNUM
EOS_NOTRDY
```

See Also

I_SETSTAT, SS_RELEASE

I_SETSTAT, SS_SIZE

Set File Size (RBF, PIPE, PCF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_size_pb {
   u_int32 filesize;
}  ss_size_pb, *Ss_size_pb;
```

Description

`SS_SIZE` sets the size of the file associated with the open path to the specified `filesize`.

If the specified size is smaller than the current size, the data beyond the new end-of-file is lost.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`filesize`	is the new size of the file in bytes.

Possible Errors

`EOS_BPNUM`

See Also

`I_SETSTAT`

I_SETSTAT, SS_SKIP

Skip Blocks (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_skip_pb {
   int32 blks;
}  ss_skip_pb, *Ss_skip_pb;
```

Description

SS_SKIP skips the specified number of blocks.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

blks	specifies the number of blocks to skip. If blks is negative, the tape is rewound the specified number of blocks.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_SETSTAT, SS_SKIPEND

Skip to End of Tape (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

This call does not use a substructure to the set status parameter block.

Description

`SS_SKIPEND` skips the tape to the end of data. This enables you to append data to tapes on cartridge-type tape drives.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Possible Errors

```
EOS_BPNUM
EOS_NOTRDY
```

See Also

`I_SETSTAT`

I_SETSTAT, SS_TICKS

Wait Specified Number of Ticks for Record Release (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_ticks_pb {
   u_int32 delay;
}  ss_ticks_pb, *Ss_ticks_pb;
```

Description:

Normally, if a read or write request is issued for part of a file locked out by another user, RBF sleeps indefinitely until the conflict is removed. `SS_TICKS` may be used to return an error (`EOS_LOCK`) to the user program if the conflict still exists after the specified number of ticks have elapsed.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`delay` specifies the delay interval. The delay interval is used directly as a parameter to RBF's conflict sleep request.

Value	Description
0	The process sleeps until the record is released. This is RBF's default.
1	Returns an error if the record is not released immediately.
Other	Any other value specifies number of system clock ticks to wait until the conflict area is released. If the high order bit is set, the lower 31 bits are converted from 1/256 second to ticks before sleeping. This allows programmed delays to be independent of the system clock rate.

Possible Errors

```
EOS_BPNUM
EOS_LOCK
```

See Also

`I_SETSTAT`

I_SETSTAT, SS_WFM

Write Tape Marks (SBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_wfm_pb {
    u_int32 cnt;
}  ss_wfm_pb, *Ss_wfm_pb;
```

Description

SS_WFM writes the specified number of tape marks at the current position.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

cnt	specifies the number of tape marks to write.

Possible Errors

EOS_BPNUM

See Also

I_SETSTAT

I_SETSTAT, SS_WTRACK

Write (Format) Track (RBF)

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct ss_wtrack_pb {
    void *trkbuf,
        *ilvtbl;
    u_int32 track,
        head,
        interleave;
}  ss_wtrack_pb, *Ss_wtrack_pb;
```

Description

SS_WTRACK 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 the track number and side number are both zero. The interleave table contains byte entries of LBNs ordered to match the requested interleave offset. The path descriptor should be used with the track and side numbers to determine what density and how many blocks a certain track should have.

This function is implemented by the driver. Only super user programs are allowed to issue this command.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`trkbuf`	points to the track buffer.
`ilvtbl`	points to the interleave table. The interleave table contains byte entries of LBNs ordered to match

	the requested interleave offset.
`track`	is the track number.
`head`	is the side number.
`interleave`	is the interleave value.

Possible Errors

```
EOS_FMTERR
EOS_FORMAT
```

See Also

`I_SETSTAT`

I_SGETSTAT

GetStat Call Using System Path Number

Headers

```
#include <types.h>
#include <sg_codes.h>
```

Parameter Block Structure

```
typedef struct i_getstat_pb {
   syscb cb;
   path_id path;
   u_init16 gs_code;
   void *param_blk;
} i_getstat_pb, *I_getstat_pb;
```

Description

I_SGETSTAT is a wildcard call used to handle individual device parameters that are not uniform on all devices or are highly hardware dependent. I_SGETSTAT provides the same functionality as I_GETSTAT except the path number for I_SGETSTAT is assumed to be a system path number and not a user path number.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

cb
is the control block header.

path
is the system path number.

gs_code
is the get status code.

param_blk
points to the parameter block corresponding to the function being performed. If the get status function does not require a parameter block param_blk should be NULL.

Possible Errors

`EOS_UNKSVC`

See Also

`I_GETSTAT`
`I_SETSTAT`

I_TRANPN

Translate User Path to System Path

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_tranpn_pb {
   syscb cb;
   process_id proc_id;
   path_id user_path,
      sys_path;
}  i_tranpn_pb, *I_tranpn_pb;
```

Description

`I_TRANPN` translates a user path number to a system path number. System-state processes use this call to access the user paths (standard I/O paths).

Attributes

Operating System:	OS-9
State:	System and I/O
Threads:	Safe

Parameters

`cb` is the control block header.

`proc_id` specifies the process ID.

`user_path` specifies the user path to translate.

`sys_path` is the mapped system path.

Possible Errors

```
EOS_BPNUM
EOS_IPRCID
```

I_WRITE

Write Data to File or Device

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_write_pb {
   syscb cb;
   path_id path;
   u_char *buffer;
   u_int32 count;
}  i_write_pb, *I_write_pb;
```

Description

`I_WRITE` outputs bytes to a file or device associated with the specified path number. 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.

On RBF devices, any locked record is released.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`path`	is the specified path number for the file or device.
`buffer`	points to the data buffer.
`count`	is the number of bytes written.

Possible Errors

EOS_BMODE
EOS_BPNUM
EOS_WRITE

See Also

I_CREATE
I_OPEN
I_WRITELN

I_WRITELN

Write Line of Text with Editing

Headers

```
#include <types.h>
```

Parameter Block Structure

```
typedef struct i_writln_pb {
   syscb cb;
   path_id path;
   u_int32 count
   u_char *buffer;
}  i_writln_pb, *I_writln_pb;
```

Description

`I_WRITELN` outputs bytes to a file or device associated with the specified path number. The path must have been opened or created in write or update access modes. `I_WRITELN` writes data until it encounters a carriage return character or `count` bytes. Line editing is also activated for character-oriented devices such as terminals and printers. The line editing refers to functions such as auto line feed and null padding at end-of-line.

The number of bytes actually written (returned in `count`) 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.

On RBF devices, any locked record is released.

Attributes

Operating System:	OS-9
State:	User, System, and I/O
Threads:	Safe

Parameters

`cb`	is the control block header.
`path`	is the path number of the file or device.
`buffer`	points to the data buffer.
`count`	is the number of bytes written.

Possible Errors

EOS_BMODE
EOS_BPNUM
EOS_WRITE

See Also

I_CREATE
I_OPEN
I_WRITE

The *OS-9 Porting Guide*, the SCF Drivers (line editing) section

A Example Code

Use the examples in this section as guides for creating your own modules. These examples should not be considered the most current software. Software for your individual system may be different.

This appendix includes the following topics:

- Sysgo
- Signals: Example Program
- Alarms: Example Program
- Events: Example Program
- Semaphores: Example Program
- Subroutine Library
- Trap Handlers

SYSGO

Sysgo can be configured as the first user process started after the system start-up sequence. Its standard I/O is on the system console device.

Sysgo executes as follows:

1. Change to the CMDS execution directory on the system device.
2. Execute the start-up file (as a script) from the SYS directory on the root of the system device.
3. Fork a shell on the system console.
4. Wait for that shell to terminate and then fork it again. Unless Sysgo dies, a shell is always running on the system console.

The standard Sysgo module for disk systems cannot be used on non-disk systems, but is easy to customize.

```
/*-------------------------------------------------------------------------!
!                                                                          !
!      Copyright 1988 by Microware Systems Corporation                     !
!      Copyright 2001 by RadiSys Corporation
!                  Reproduced Under License                                !
!                                                                          !
!This source code is the proprietary confidential property of Microware    !
!Systems Corporation, and is provided to licensee for documentation and    !
!educational purposes only. Reproduction, publication, or distribution     !
!in any form to any party other than the licensee is strictly prohibited.  !
!                                                                          !
!--------------------------------------------------------------------------*/
_asm("_sysedit: equ 2");

#include    <const.h>
#include    "defsfile"

/*
 * global variables and declarations
 */

u_int32         sighandler(),              /* intercept handler */
                os9fork();                 /* used by os9exec */
void            errexit(),                 /* error printing routine */
                out3dec();                 /* print three decimal digits */
error_code      lerrmsg();                 /* print the error message */
char            *cmdsdir = "CMDS",         /* the commands directory */
                *startup = "SYS/startup",  /* the startup script */
                *shell = "Shell";          /* the shell command name */

/*
 * main - main program body
 */

void main(argc, argv)
```

```
register u_int32            argc;           /* number of arguments */
register u_char             *argv[];        /* the arguments themselves */
{
 register path_id           stdid_dup;      /* duped stdin ID */
 register process_id        shellpid;       /* the process ID */
 char                       *envp[1];       /* environment variables */
 static char                *args[] = {     /* argv for forked shell */
                                "shell",
                                "-npxt\n",
                                NULL
 };

   intercept(sighandler);                     /* catch signals */
   if (chxdir(cmdsdir) == ERROR)
       errexit(errno, "can’t change to commands directory");
   if ((stdid_dup = dup(_fileno(stdin))) == ERROR)
       errexit(errno, "can’t duplicate standard input path");
   close(_fileno(stdin));                      /* close stdin path */
   if (open(startup, S_IREAD) == ERROR) {
       lerrmsg(errno, "can’t open startup due to error #");
       dup(stdid_dup);                        /* reset stdin path */
   }
   envp[0] = NULL;                            /* initialize environments */
   for (;;) {
       if (os9exec(os9fork, shell, args, envp, 0, 0, 3) == ERROR)
          errexit(errno, "can’t fork shell");
       close(_fileno(stdin));                 /* close old stdin */
       dup(stdid_dup);                        /* restore initial stdin */
       wait(0);                               /* wait for it to die */
       args[1] = "\n";                        /* no more special options */
   }
}

/*
 * sighandler - ignore signals so we stay alive
 */

u_int32 sighandler(sigval)
register u_int32    sigval;                 /* the signal */
{
    return SUCCESS;                         /* don’t quit */
}

/*
 * errexit - print error message and leave
 */


void errexit(error, msg)
register error_code     error;            /* the error that caused us to quit
*/
register char           *msg;               /* our explanation */
{
    write(_fileno(stdout), msg, strlen(msg));
    exit(lerrmsg(error, " due to error #"));
}

/*
```

```
 * lerrmsg - print error message and number
 */

error_code lerrmsg(error, msg)
register error_code     error;              /* the error code */
register char           *msg;               /* the error message */
{
    write(_fileno(stdout), msg, strlen(msg));
    out3dec(error >> 16);
    write(_fileno(stdout), ":", 1);
    out3dec(error & 0xffff);
    writeln(_fileno(stdout), "\n", 1);
}

/*
 * out3dec - output 3 decimal digits
 */

void out3dec(num)
register u_int32        num;               /* the number to print */
{
 register u_int32       i,                 /* a counter */
                        j;                 /* divisor */
 char                  buf[3];            /* the buffer for the characters */

 for (i = 0, j = 100; i < 3; i++, j /= 10)
     buf[i] = (num / j) + 0x30;            /* convert to decimal */
 write(_fileno(stdout), buf, 3);
}
```

SIGNALS: EXAMPLE PROGRAM

The following program demonstrates a subroutine that reads a `\n` terminated string from a terminal with a ten second timeout between the characters. This program illustrates signal usage, but does not contain any error checking.

The `_ss_ssig(path, value)` library call notifies the 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 is returned 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 that 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) ) {
          _os_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;
     }
}
```

ALARMS: EXAMPLE PROGRAM

The following example program can be compiled 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>
#include <const.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 ((errno = _os_alarm_set(&alarm_id, 2, TIME(secs))) != SUCCESS)
        exit(_errmsg(errno, "can’t set alarm - "));

   if (gets(name) != 0)
        _os_alarm_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, complete the following steps:

Step 1. Enter or copy 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 using this command: `sema1 & sema2`.

The program does the following:

1. Creates an event with an initial value of `1` (free), a wait increment of `-1`, and a signal increment of `1`
2. Enters a loop that waits on the event
3. Prints a message
4. Sleeps
5. Signals the event
6. Unlinks itself from the event after ten times through the loop
7. Deletes the event from the system

```
#include <module.h>
#include <stdio.h>
#include <memory.h>
#include <errno.h>
#include <const.h>

void main()
{
    char         *ev_name = "semaevent";   /* name of event to be used */
    event_id    ev_id;                  /* ID that is used to access event */
    u_int16      perm = MP_OWNER_READ | MP_OWNER_WRITE; /* access perms for
event */
    u_int32      value;                    /* returned event value */
    signal_code  signal;                   /* returned signal value */
    int          count = 0;                /* loop counter */

    /* create to link to the event */
    if (( errno = _os_ev_link(ev_name, &ev_id)) != SUCCESS)
        if ((errno = _os_ev_creat(1,-1,perm,&ev_id,ev_name,1,MEM_ANY)) !=
SUCCESS)
            exit(_errmsg(errno,"error getting access to event - "));

    while (count++ < 10)
    {
        /* wait on the event */
        if ((errno = _os_ev_wait(ev_id, &value, &signal, 1, 1)) != SUCCESS)
        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 ((errno = _os_ev_signal(ev_id, &value, 0)) != SUCCESS)
            exit(_errmsg(errno, "error signalling the event -"));
        /* simulate doing something other than critical section */
        sleep(1);
    }
    /* unlink from event */
    if ((errno = _os_ev_unlink(ev_id)) != SUCCESS)
        exit(_errmsg(errno, "error unlinking from event - "));

    /* delete event from system if this was the last process to unlink from it
*/
    if ((errno = _os_ev_delete(ev_name)) != SUCCESS && errno != EOS_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.

```
#include <stdio.h>
#include <stdlib.h>
#include <types.h>
#include <module.h>
#include <modes.h>
#include <semaphore.h>

#define DMNAME “hplaserjet”

semaphore *printerSema;

error_code main(int, char **);
error_code printfile(char *);

error_code main(int argc, char **argv){
     mh_data *modptr;
     u_int16 attrev, typlang;
     u_int32 perm;
     char *dm_name = DMNAME;
     error_code error;

     /* prepare parameters */
     attrev = mkattrevs(MA_REENT,0);
     typlang = mktypelang(MT_DATA,ML_ANY);
     perm = MP_OWNER_READ|MP_OWNER_WRITE|
            MP_GROUP_READ|MP_GROUP_WRITE;
     /* first try to create the module */
     error = _os_datmod(dm_name, sizeof(*printerSema),
                        &attrev, &typlang,perm,
                        (void**)&printerSema, &modptr);

     if(error){
          /* then try to link to it */
          error = _os_link(&dm_name, (mh_com**)&modptr,
                           (void**)&printerSema, &typlang,
                           &attrev);
          if(error){
               fprintf(stderr,”Couldn’t link or create! Error#%d\n”,error);
               _os_exit(error);
          }
     }
     /* initilize semaphore */
     _os_sema_init(printerSema);

     /* compete for the resource */
     _os_sema_p(printerSema);

     /* print the file */
     printfile(argv[1]);

     /* release the semaphore */
     _os_sema_v(printerSema);


     /* terminate semaphore */
     _os_sema_term(printerSema);
     /* unlink the data module */
     _os_unlink(modptr);

     _os_exit(0);
}

#define PRINTER “/p”
#define BUFSIZE 256
error_code printfile(char *filename){
     path_id prnpath, dskpath;
     char buffer[BUFSIZE];
     u_int32 size;
     error_code error;

     /* open path to printer */
     error = _os_open(PRINTER, FAM_WRITE, &prnpath);
     if(error != 0){
          return(error);
     }

     /* open path to disk file */
     error = _os_open(filename, FAM_READ, &dskpath);
     if(error != 0){
          _os_close(prnpath);
          return(error);
     }

     /* until the last byte, read from disk and write to printer */
     do{
          size = BUFSIZE;
          error = _os_read(dskpath, buffer, &size);
          if(error == 0 && size > 0){
               _os_write(prnpath, buffer, &size);
          }
     }while(size > 0);

     _os_close(dskpath);
     _os_close(prnpath);
     return(0);
}
```

SUBROUTINE LIBRARY

Subroutine Module

Below is an example of a subroutine module with two entrypoints. The subroutine module consists of two files: the root psect file (in assembly language), and the subroutine module function file (in C language).

root psect file

```
*substart.a: Generic psect for OS-9 subroutine modules use
<oskdefs.d>

Edition equ 1

Typ_Lang set (Sbrtn<<8)+Objct
Attr_Rev set ((ReEnt+SupStat)<<8)+0
psect hcstart_a,Typ_Lang,Attr_Rev,Edition,0,0,exec
* This portion lists the functions that exist in the
subroutine module.
* This is essentially an array of function pointers.
exec:
 dc.l sub_init-btext
 dc.l sub_function_1-btext
 dc.l 0
 ends
```

function file

```
/*
** This is a "C" file that implements sub_init and   **
sub_function_1. Remember to turn of stack checking ** (-r) and compile
with -bepg. Global variables are ** ok to access in this file, since
the calling     ** routine will set up your global pointer.
*/
error_code sub_init(int init_param)
{
...
    return SUCCESS;
}
error_code sub_function_1(u_int32 *ticks)
{
    *ticks = ...;

    return SUCCESS;
}
```

Application Call into a Subroutine Module

This section describes how an application calls into a subroutine library. It also describes how to initialize a subroutine module and how to discontinue using it.

Initialization

Before you can begin using a subroutine library, you must first link to it using the `_os_slink` call, as several values returned from `_os_slink` must be stored. Below is an example of how to link to a subroutine module and store the necessary returned values.

```
#include <module.h>
#include <types.h>
#include <errno.h>
#include <stdlib.h>
#include <regs.h>

typedef struct {
  u_int32 *exec;     /* execution point for hcsub */
  void    *gp;   /* global (static storage) pointer for hcsub */
  mh_com  *mod_head; /* module header */
  int     subnum;    /* subroutine number (0..15) */
} SUB_DATA;

/* defines array index of subroutine module function */
#define SUB_INIT 0
#define SUB_FUNCTION_1 1

/* this links to the subroutine module called 'mysub' and stores ** the module
head, execution offset, global pointer, and      ** subroutine number in the
SUB_DATA structure.*/
error_code _sub_link(SUB_DATA *hc, int subnum)
{
  hc->subnum = subnum;
  return _os_slink(subnum,"mysub",(void**)&hc->exec,&hc->gp,&hc->mod_head);
}

/* this unlinks from the subroutine module.  Only call this when ** you are
all done using 'mysub'.
*/
error_code _sub_unlink(SUB_DATA *hc)
{
  void *ptr;
  error_code err;

  err = _os_slink(hc->subnum,NULL,&ptr,&ptr,(mh_com**)&ptr);

  return err;
}
Calling into the subroutine module: The following shows functions that setup
and call into the subroutine module.  These functions may be placed into a
library if several applications are to access the subroutine module.
```

```
/*
** This calls the 'sub_init' entrypoint into the subroutine    ** module. This
will pass one parameter to the init function: **'init_param'.
*/
error_code _sub_init(SUB_DATA *hc, int init_param)
{
  error_code (*func)();
  error_code    err;
  void          *oldstatics;

 func = (error_code(*)())((u_int8*)hc->mod_head + hc->exec[SUB_INIT]);
oldstatics = change_static(hc->gp);
err = func();
(void) change_static(oldstatics);

  return err;
}

/*
** This calls into the "sub_function_1' entrypoint of the      ** subroutine
module.

** This passes one parameter:  'ticks'.
*/
error_code _sub_function_1(SUB_DATA *hc, u_int32 *ticks)
{
  error_code (*func)(u_int32*);
  error_code    err;
  void          *oldstatics;

  func = (error_code(*)(u_int32*))((u_int8*)hc->mod_head + hc-
>exec[SUB_FUNCTION_1]);
  oldstatics = change_static(hc->gp);
  err = func(ticks);
  (void) change_static(oldstatics);

  return err;
}
```

TRAP HANDLERS

The following example trap handler consists of four files: `trapc.a`, `thandler.c`, `tcall.c`, and `ttest.c`.

trapc.a

```
****************************************************************************
*                                                                          *
*              Copyright 1989 by Microware Systems Corporation             *
*              Copyright 2001 by RadiSys Corporation                       *
*                          Reproduced Under License                        *
*                                                                          *
*  This source code is the proprietary confidential property of Microware  *
*  Systems Corporation, and is provided to licensee for documentation and  *
*  educational purposes only. Reproduction, publication, or distribution   *
*  in any form to any party other than the licensee is strictly prohibited. *
*                                                                            *
******************************************************************************

 nam OS-9000 80386 Example System State Trap Handler

 use <oskdefs.d>

type equ (TrapLib<<8)+Objct
revs equ ((ReEnt+Ghost+SupStat)<<8)
edit equ 1
stack equ 1024

 psect Trap_9000,type,revs,edit,stack,_trap_entry
 _m_init: equ _trap_init    * Trap Handler initialization entry point
 _m_term: equ _trap_term    * Trap Handler termination entry point

_sysedit: equ edit edition number of module

E_ILLFNC equ $40    Illegal trap handler function code error

 vsect
_caller_eip: ds.l 1         caller's return pc
_caller_statics: ds.l 1     caller's static storage pointer (%ebx)
 ends

******************************************************************************
*
* _trap_entry - trap handler entry point code.
*
*   input:  0(%esp) = caller's static storage pointer (%ebx)
*           4(%esp) = trap number
*           6(%esp) = function code
*           8(%esp) = return address
*
_trap_entry: push.l %eax save registers
 push.l %esi
stacked set 2*4
 sub.l %eax,%eax sweep register
 mov.w 6+stacked(%esp),%eax get function code
 cmp.l trap_max(%ebx),%eax function code in range?
 jge.b _bad_trap branch if not
 lea trap_dsptable(%ebx),%esi get trap dispatch table
 mov.l (%esi,%eax*4),%eax get routine address
 mov.l %eax,4+stacked(%esp) set routine address
 pop.l %esi restore registers
 pop.l %eax
 pop.l _caller_statics(%ebx) save caller's static storage
* call trap handler function
 ret

_bad_trap  pop.l %esi restore registers
 pop.l %eax
 lea 2*4(%esp),%esp pop stack
 mov.l #E_ILLFNC,%eax return error code
 ret

 ends
```

thandler.c

```
/*--------------------------------------------------------------------------,
!                                                                           !
!              Copyright 1989 by Microware Systems Corporation              !
!              Copyright 2001 by RadiSys Corporation                        !
!                          Reproduced Under License                         !
!                                                                           !
!  This source code is the proprietary confidential property of Microware   !
!  Systems Corporation, and is provided to licensee for documentation and   !
!  educational purposes only. Reproduction, publication, or distribution    !
!  in any form to any party other than the licensee is strictly prohibited. !
!                                                                           !
‘--------------------------------------------------------------------------*/
/*
    System State Trap Handler Example.  This file contains the trap handler
    dispatch table and functions.
*/

#include <const.h>

/* pre-declare trap handler functions */
int func1(), func2(), func3();

/* initialize maximum function count variable */
int trap_max = 3;

/* initialize trap handler dispatch table */
(* trap_dsptable[])() =  {
    func1,
    func2,
    func3
};


/* _trap_init - trap handler initialization routine. */
_trap_init(trapnum, memsize, statics)
register int trapnum;           /* trap handler number */
register int memsize;           /* addtional trap handler memory size */
register void *statics;         /* caller’s static storage pointer */
{
    return SUCCESS;
}

/* _trap_term - trap handler termination routine. */
_trap_term(trapnum, statics)
register int trapnum;           /* trap handler number */
register void *statics;         /* caller’s static storage pointer */
{
    return SUCCESS;
}


/* func1 - first trap handler function. */
func1()
{
    return 1;
}
```

```
/* func2 - second trap handler function. */
func2()
{
    return 2;
}

/* func3 - third trap handler function. */
func3()
{
    return 3;
}
```

tcall.c

```
/*--------------------------------------------------------------------------,
!                                                                           !
!              Copyright 1989 by Microware Systems Corporation              !
!              Copyright 2001 by RadiSys Corporation                        !
!                          Reproduced Under License                         !
!                                                                           !
!  This source code is the proprietary confidential property of Microware   !
!  Systems Corporation, and is provided to licensee for documentation and   !
!  educational purposes only. Reproduction, publication, or distribution    !
!  in any form to any party other than the licensee is strictly prohibited. !
!                                                                           !
'-------------------------------------------------------------------------*/
/*
    Example system state trap handler calls for 80386 processor.  This file
    contains the tcall references for the trap handler functions.  The main
    program references these tcalls, and in turn the tcalls will dispatch
    to the associated trap handler via the OS9000 kernel.  The return from
    the trap handler takes the flow of excution back to the initial function
    reference in the main program.
*/

_asm ("

***************************
* tcall - macro definition
*
* tcall trap,function
*
tcall macro
 dc.w $fecd
 dc.w \1
 dc.w \2
 ret
 dc.b $00
 endm

trap_func1: tcall 8,0
trap_func2: tcall 8,1
trap_func3: tcall 8,2

");
```

ttest.c

```
/*
    System State Trap Handler test program.
*/

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

#ifndef SUCCESS
#define SUCCESS 0
#endif

char *libexec;
char *modhead;

/* _trapinit - trap handler exception routine, install trap handler. */
_trapinit(trapnum, funcode)
register int trapnum;
register int funcode;
{
    register int err;

    /* validate trap number */
    if (trapnum != 8)  return errno = EOS_ITRAP;

    /* install the trap handler */
    if ((err = _os_tlink(8, "trap9000", &libexec, &modhead, 0, 0)) != SUCCESS)
        return errno = err;

    return SUCCESS;
}


main()
{
    printf("calling function %d.\n", trap_func1());
    printf("calling function %d.\n", trap_func2());
    printf("calling function %d.\n", trap_func3());
}
```

B OS-9 Error Codes

This section lists OS-9 error codes in numerical order. The first three numbers indicate a group of messages. Processor-specific error messages can also be added with each processor family port. If this manual has not been updated to include the messages for your processor, see the `errmsg` file in the `OS9000/SRC/SYS/ERRMSG` directory. This appendix includes the following topics:

- Error Categories
- Errors

ERROR CATEGORIES

OS-9 error codes are grouped in the following categories:

Table B-1.OS-9 Error Code Categories

Range	Description
000:001 - 000:031	**Miscellaneous Errors** Refer to Table B-2.
000:032 - 000:047	**Ultra C Related Errors** Refer to Table B-3.
000:060 - 000:069	**Miscellaneous Program Errors** Refer to Table B-4.
000:080 - 000:089	**Miscellaneous OS Errors** Refer to Table B-5.
000:102 - 000:132 000:134 - 000:163	**Reserved Errors** Refer to Table B-6.
000:133	Uninitialized User Trap (1-15) Error Refer to Table B-6.
000:164 - 000:239	**Operating System Errors** These errors are normally generated by the kernel or file managers. Refer to Table B-7.
000:240 - 000:255	**I/O Errors** These error codes are generated by device drivers or file managers. Refer to Table B-8.
000:256	**ANSI C Errors** ANSI C math out of range error Refer to Table B-9.
001:000 - 001:099	**Compiler Errors** Refer to Table B-10.
006:100 - 006:206	**RAVE Errors.** Call Microware Customer Support for more information. Refer to Table B-11.
007:001 - 007:029	**Internet Errors** Refer to Table B-12.

Table B-1.OS-9 Error Code Categories

Range	Description
008:257 - 008:288	IEEE1394 Errors Refer to Table B-13.
020:001 - 020:022	POSIX Errors Refer to Table B-14.
100:000 - 100:999	**PowerPC Processor-specific Errors** Refer to Table B-15.
102:000 - 102:032	**MIPS Processor-specific Errors** Refer to Table B-16.
103:000 - 103:008	**ARM Processor-specific Errors** Refer to Table B-17.

ERRORS

The following OS-9 error codes are defined in the `errno.h` file.

Table B-2.Miscellaneous Error Codes

Number	Name	Description
000:001		Process has aborted.
000:002	`S_Abort signal`	Keyboard quit (^E) typed.
000:003	`S_Intrpt signal`	Keyboard interrupt (^C) typed.
000:004	`S_HangUp signal`	Modem hangup.

Table B-3.Ultra C Error Codes

Number	Name	Description
000:032	`EOS_SIGABRT`	An abort signal was received.
000:033	`EOS_SIGFPE`	An erroneous math operation signal was received.
000:034	`EOS_SIGILL`	An illegal function image signal was received.
000:035	`EOS_SIGSEGV`	A segment violation (bus error) signal was received.
000:036	`EOS_SIGTERM`	A termination request signal was received.

Table B-3.Ultra C Error Codes

Number	Name	Description
000:037	EOS_SIGALRM	An alarm time elapsed signal was received.
000:038	EOS_SIGPIPE	A write to pipe with no readers signal was received.
000:039	EOS_SIGUSR1	A user signal #1 was received.
000:040	EOS_SIGUSR2	A user signal #2 was received.
000:041	EOS_SIGCHECK	A machine check exception signal was received.
000:042	EOS_SIGALIGN	An alignment exception signal was received.
000:043	EOS_SIGINST	An instruction access exception signal was received.
000:044	EOS_SIGPRIV	A privilege violation exception signal was received.

Table B-4.Miscellaneous Program Error Codes

Number	Name	Description
000:064	EOS_ILLFNC	Illegal function code.
000:065	EOS_FMTERR	ASCII to numeric format conversion error.
000:066	EOS_NOTNUM	Number not found.
000:067	EOS_ILLARG	Illegal argument.
000:067	EINVAL	Invalid argument (POSIX).

Table B-5.Miscellaneous Operating System Error Codes

Number	Name	Description
000:080	EOS_MEMINUSE	Memory already in use.
000:081	EOS_UNKADDR	Do not know how to translate.

Table B-6.Operating System Reserved Error Codes

Number	Name	Description
000:102	`EOS_BUSERR`	A bus trap error occurred.
000:103	`EOS_ADRERR`	An address trap error occurred.
000:104	`EOS_ILLINS`	An illegal instruction exception occurred.
000:105	`EOS_ZERDIV`	A zero divide exception occurred.
000:106	`EOS_CHK`	A `chk` or `chk2` instruction trap occurred.
000:107	`EOS_TRAPV`	A `trapv` or `trapcc` instruction occurred.
000:108	`EOS_VIOLAT`	A privileged instruction violation occurred.
000:109	`EOS_TRACE`	An uninitialized Trace exception occurred.
000:110	`EOS_1010`	A 1010 instruction exception occurred.
000:111	`EOS_1111`	A 1111 instruction exception occurred.
000:112	`EOS_RESRVD`	An invalid exception occurred (#12).
000:113	`EOS_CPROTO`	Coprocessor protocol violation.
000:114	`EOS_STKFMT`	System stack frame format error.
000:115	`EOS_UNIRQ`	An uninitialized interrupt occurred.
000:116 - 000:123		An invalid exception occurred (#16 - #23).
000:124		Spurious Interrupt occurred.
000:133	`EOS_TRAP`	An uninitialized user TRAP (1-15) was executed.
000:148	`EOS_FPUNORDC`	Floating point coprocessor unordered condition.
000:149	`EOS_FPINXACT`	Floating point coprocessor inexact result.
000:150	`EOS_FPDIVZER`	Floating point coprocessor divide by zero.
000:151	`EOS_FPUNDRFL`	Floating point coprocessor underflow.
000:152	`EOS_FPOPRERR`	Floating point coprocessor operand error.
000:153	`EOS_FPOVERFL`	Floating point coprocessor overflow.
000:154	`EOS_FPNOTNUM`	Floating point coprocessor not a number.
000:155	`EOS_FPUNDATA`	FP Unsupported data type.
000:156	`EOS_MMUCONF`	PMMU Configuration exception.

Table B-6.Operating System Reserved Error Codes (Continued)

Number	Name	Description
000:157	EOS_MMUILLEG	PMMU Illegal Operation exception.
000:158	EOS_MMUACCES	PMMU Access Level Violation exception.
000:159 - 000:163		An invalid exception occurred (#59 - #63).

Table B-7.OS-9-Specific Error Codes

Number	Name	Description
000:164	EOS_PERMIT	No permission. A user process has attempted something that can only be done by a system *super user*.
000:164	EACESS	POSIX access denied.
000:165	EOS_DIFFER	The arguments to F_CHKNAM do not match.
000:166	EOS_STKOVF	System stack overflow. F_ChkNam can return this error if the pattern string is too complex.
000:167	EOS_EVNTID	Invalid or Illegal event ID number specified.
000:168	EOS_EVNF	Event name not found.
000:169	EOS_EVBUSY	The event is busy (and can’t be deleted).
000:170	EOS_EVPARAM	Impossible event parameters supplied.
000:171	EOS_DAMAGE	System data structures have been damaged.
000:172	EOS_BADREV	Module revision is incompatible with operating system.
000:173	EOS_PTHLOST	Path became lost because network node was down.
000:174	EOS_BADPART	Bad (disk) partition data, or no active partition.
000:175	EOS_HARDWARE	Hardware damage has been detected.
000:176	EOS_NOTME	Not my device. Error returned by an interrupt service routine when it is polled for an interrupt its device did not cause.

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:177	`EOS_BSIG`	Fatal signal or no intercept routine. Process received a fatal signal or did not have an intercept function.
000:178	`EOS_BUF2SMALL`	The buffer passed is too small. A routine was passed a buffer too small to hold the data requested.
000:179	`EOS_ISUB`	Illegal/used subroutine module number.
000:180	`EOS_EVTFUL`	Event descriptor table full.
000:196	`EOS_SYMLINK`	Symbolic link found in path list. A link was attempted that would have caused recursion in the file structure. You may not link to a directory containing the real directory.
000:197	`EOS_EOLIST`	End of alias list.
000:198	`EOS_LOCKID`	Illegal I/O lock identifier specified. Usually this error occurs because a user has initialized a device for use with more than one file manager.
000:199	`EOS_NOLOCK`	Lock not obtained.
000:200	`EOS_PTHFUL`	The user’s (or system) path table is full. Usually this error occurs because a user program has tried to open more than 32 I/O paths simultaneously. It might also occur if the system path table becomes full and can not be expanded.
000:201	`EOS_BPNUM`	Bad path number. An I/O request has been made with an invalid path number, or one not currently open.
000:201	`EBADF`	Bad file descriptor (POSIX).
000:202	`EOS_POLL`	The system IRQ table is full. To install another interrupt producing device, one must first be removed. The system’s `init` module specifies the maximum number of IRQ devices that may be installed.

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:203	EOS_BMODE	Bad I/O mode. An attempt has been made to perform I/O on a path incapable of supporting it. For example, writing to a path open for input.
000:204	EOS_DEVOVF	The system's device table is full. To install another device descriptor, one must first be removed. The system init module can be changed to allow more devices.
000:205	EOS_BMID	Bad module ID. An attempt has been made to load a module without a valid module header.
000:206	EOS_DIRFUL	The module directory is full. No more modules can be loaded or created unless one is first unlinked. Although OS-9 automatically expands the module directory when it becomes full, this error may be returned because the there is not enough memory or the memory is too fragmented to use.
000:207	EOS_MEMFUL	Memory full. This error is returned from the `F_SRqMem` service call when there is not enough system RAM to fulfill the request, or if a process has already been allocated the maximum number of blocks permitted by the system.
000:208	EOS_UNKSVC	Unknown service code. An OS-9 call specified an unknown or invalid service code, or a getstat/setstat call was made with an unknown status code.
000:209	EOS_MODBSY	The module is busy. An attempt has been made to access (through `F_Link`) a non-sharable module or non-sharable device already in use.

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:210	`EOS_BPADDR`	Bad page address. A memory de-allocation request has been given a buffer pointer or size that is invalid, often because it references memory that has not been allocated to the caller. The system detects trouble when the buffer is returned to free memory or if it is used as the destination of a data transfer, such as `I_Read`.
000:210	`EFAULT`	Bad address (POSIX).
000:211	`EOS_EOF`	The end of file has been reached. An end of file condition was encountered on a read operation.
000:211	`EPIPE`	Broken pipe (POSIX).
000:212	`EOS_VCTBSY`	IRQ vector is busy. A device has tried to install itself in the IRQ table to handle a vector claimed by another device.
000:213	`EOS_NES`	Non-existing segment. A search was made for a disk file segment that cannot be found. The device could have a damaged file structure.
000:214	`EOS_FNA`	File not accessible. An attempt to open a file failed. The file was found, but is inaccessible in the requested mode. Check the file’s owner ID and access attributes.
000:214	`EPERM`	Operation not permitted (POSIX).
000:215	`EOS_BPNAM`	Bad pathlist specified. The specified pathlist has a syntax error, for example, an illegal character.
000:216	`EOS_PNNF`	File not found. The specified pathlist does not lead to any known file.
000:216	`ENOENT`	No such file or directory (POSIX).

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:217	`EOS_SLF`	File segment list is full. A file has become too fragmented to accommodate further growth. This can occur on a nearly full disk, or one whose free space has become scattered. The simplest way to solve the problem is to copy the file, which should move it into more contiguous space.
000:218	`EOS_CEF`	Tried to create an existing file. The specified filename already appears in the current directory.
000:218	`EEXIST`	File exists (POSIX).
000:219	`EOS_IBA`	Illegal memory block specified. The system was called to return memory, but was passed an invalid pointer or block size.
000:220	`EOS_HANGUP`	Telephone (modem) connection terminated. This error is returned when an I/O operation is attempted on a path after irrecoverable line problems have occurred, such as data carrier lost. It may be returned from network devices, if the network connection is lost.
000:221	`EOS_MNF`	Module not found. An `F_Link` call was made to a module not in memory. Modules with corrupted or modified headers will not be found.
000:222	`EOS_NOCLK`	No system clock. A request was made requiring a system clock, but one is not running. For example, a timed `F_Sleep` call has been requested, but the clock was not running. The `setime` utility is used to start the system clock.
000:223	`EOS_DELSP`	Deleting stack memory. A process tried to return the memory containing it's current stack pointer to the system. This is also known as a suicide attempt.

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:224	`EOS_IPRCID`	Illegal process ID. A system call was passed a process ID to a non-existent or inaccessible process.
000:224	`ESRCH`	No such process (POSIX).
000:225	`EOS_PARAM`	Bad parameter. A system call was passed an illegal or impossible parameter.
000:226	`EOS_NOCHLD`	No children. An `F_Wait` call was made with no child processes to wait for.
000:226	`ECHILD`	No child process (POSIX)
000:227	`EOS_ITRAP`	Invalid trap number. An `F_Tlink` call was made with an invalid user trap code or one already in use.
000:228	`EOS_PRCABT`	The process has been aborted.
000:229	`EOS_PRCFUL`	Too many active processes. The system's process table is full. (Too many processes are currently running.) The kernel automatically tries to expand the process table, but returns this error if there is not enough contiguous memory to do so.
000:230	`EOS_IFORKP`	Illegal fork parameter (not currently used)
000:231	`EOS_KWNMOD`	Known module. A call was made to install a module that is already in memory.
000:232	`EOS_BMCRC`	Bad module CRC. A CRC calculation is performed on every module when it is installed in the system module directory. Only modules with good CRCs are accepted. To generate a valid CRC value in an intentionally altered module, use the `fixmod` utility.
000:233	`EOS_SIGNAL`	Signal error (replaces `EOS_USIGP`.)
000:234	`EOS_NEMOD`	Non executable module.
000:235	`EOS_BNAM`	Bad name. This error is returned by the `F_PrsNam` system call if there is a syntax error in the name.

Table B-7.OS-9-Specific Error Codes (Continued)

Number	Name	Description
000:236	`EOS_BMHP`	Bad module header parity.
000:237	`EOS_NORAM`	No RAM available. A process has made an `F_Mem` request to expand its memory size. `F_Mem` is no longer supported and `F_SrqMem` should be used. This error may also be returned if there is not enough contiguous memory to process a fork request or if a device driver does not specify any static storage requirements.
000:237	`ENOMEM`	Insufficient RAM (POSIX).
000:238	`EOS_DNE`	The directory is not empty. The directory attribute of a file cannot be removed unless the directory is empty. This prevents accidental loss of disk space.
000:239	`EOS_NOTASK`	No available task number. All of the task numbers are currently in use and a request was made to execute or create a new task. This error could be returned by a system security module (SSM).

Table B-8.OS-9-Specific I/O Error Codes

Number	Name	Description
000:240	`EOS_UNIT`	Illegal unit (drive) number.
000:241	`EOS_SECT`	Bad disk sector number.
000:242	`EOS_WP`	Media is write protected.
000:243	`EOS_CRC`	Bad module cyclic redundancy check value. A CRC error occurred on read or write verity.
000:244	`EOS_READ`	Read error. A data transfer error occurred during a disk read operation, or an SCF (terminal) input buffer overrun.
000:244	`EIO`	POSIX I/O error.

Table B-8.OS-9-Specific I/O Error Codes (Continued)

Number	Name	Description
000:245	`EOS_WRITE`	Write error. A hardware error occurred during a disk write operation.
000:246	`EOS_NOTRDY`	Device not ready.
000:246	`ENODEV`	No such device (POSIX).
000:246	`ENXIO`	No such device or address (POSIX).
000:247	`EOS_SEEK`	Seek error. A physical seek operation was unable to find the specified sector.
000:248	`EOS_FULL`	Media full. Media has insufficient free space.
000:249	`EOS_BTYP`	Bad type (incompatable media). A read operation was attempted on incompatible media. For example, a read operation for a double-sided disk was tried on a single-sided disk.
000:250	`EOS_DEVBSY`	Device busy. A non-sharable device is in use.
000:250	`EBUSY`	POSIX resource busy.
000:251	`EOS_DIDC`	Disk ID change. RBF copies the disk ID number (from sector zero) 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 zero is read; it is therefore possible to swap disks without RBF noticing. This check helps to prevent this possibility.
000:252	`EOS_LOCK`	Record is busy. Another process is accessing the record. Normal record locking routines wait forever for a record in use by another user to become available. However, RBF may be told (through a `SetStat` call) to wait for a finite amount of time. If the time expires before the record becomes free, this error is returned.

Table B-8.OS-9-Specific I/O Error Codes (Continued)

Number	Name	Description
000:253	EOS_SHARE	Non-sharable file/device is 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 file. This error is commonly returned when an attempt is made to delete an open file.
000:254	EOS_DEADLK	I/O deadlock error. This error is returned when two or more processes are waiting for each other to release I/O resources before they can proceed. One process must release control to enable the other to proceed.
000:255	EOS_FORMAT	Device is format protected. This error occurs when an attempt is made to format a format protected disk. A bit in the device descriptor may be changed to allow the device to be formatted. Formatting is usually inhibited on hard disks to prevent accidental erasure.

Table B-9.OS-9-Specific ANSI C Error Codes

Number	Name	Description
000:256	ERANGE	ANSI C math out of range error.

Table B-10.OS-9-Specific Compiler Error Codes

Number	Name	Description
001:000	ERANGE	ANSI C Number out of range error.
001:001	EDOM	ANSI C Number Not in Domain.

Table B-11.OS-9-Specific RAVE Error Codes

Number	Name	Description
006:000	`EOS_ILLPRM`	Illegal parameter. An illegal parameter was passed to a function.
006:001	`EOS_IDFULL`	Identifier (ID) table full. An ID table could not be expanded any further.
006:002	`EOS_BADSIZ`	Bad size error.
006:003	`EOS_RGFULL`	Region definition full (overflow). The region is too complex.
006:004	`EOS_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	`EOS_NULLRG`	Null region.
006:006	`EOS_BADMOD`	Bad drawmap/pattern mode. An illegal mode was passed to create a drawmap or pattern.
006:007	`EOS_NOFONT`	No active font. No font was activated when an attempt to output text was made.
006:008	`EOS_NODM`	No drawmap. No character output drawmap was available when attempting an `_os_write` or `_os_writeln` call.
006:009	`EOS_NOPLAY`	No audio play in progress. An attempt was made to stop an audio play when none was in progress.
006:010	`EOS_ABORT`	Asynchronous operation aborted.
006:011	`EOS_QFULL`	Audio queue is full. The driver queue could not handle the number of soundmaps you were attempting to output.
006:012	`EOS_BUSY`	Audio processor is busy.

Table B-12.OS-9-Specific Internet Error Codes

Number	Name	Description
007:001	EWOULDBLOCK	I/O operation would block. An operation was attempted that would cause a process to block on a socket in non-blocking mode.
007:001	EAGAIN	POSIX item temporarily available.
007:002	EINPROGRESS	I/O operation now in progress. An operation taking a long time to complete was performed, such as a `connect()` call, on a socket in non-blocking mode.
007:003	EALREADY	Operation already in progress. An operation was attempted on a non-blocking object that already had an operation in progress.
007:003	EINTR	Interrupted function call (POSIX).
007:004	EDESTADDRREQ	Destination address required. The attempted socket operation requires a destination address.
007:005	EMSGSIZE	Message too long. A message sent on a socket was larger than the internal message buffer or some other network limit.
007:006	EPROTOTYPE	Protocol wrong type for socket. A protocol was specified that does not support the semantics of the socket type requested.
007:007	ENOPROTOOPT	Bad protocol option. A bad option or level was specified in a `getsockopt()` or `setsockopt()` call.
007:008	EPROTONOSUPPORT	Protocol not supported. The requested protocol is not available or not configured for use.
007:009	ESOCKNOSUPPORT	Socket type not supported. The requested socket type is not supported or not configured for use.
007:010	EOPNOTSUPP	Operation unsupported on socket.
007:011	EPFNOSUPPORT	Protocol family not supported.

Table B-12.OS-9-Specific Internet Error Codes (Continued)

Number	Name	Description
007:012	EAFNOSUPPORT	Address family unsupported by protocol.
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 an attempt is made to create a socket with an address not on the local machine.
007:015	ENETDOWN	Network is down.
007:016	ENETUNREACH	Network is unreachable. This is usually caused by network interface hardware that is operational, but not physically connected to the network. This error is also returned when the network has no way to reach the destination address.
007:017	ENETRESET	Network lost connection on reset. The host crashed and rebooted.
007:018	ECONNABORTED	Software caused connection abort. The local (host) machine caused a connection abort.
007:019	ECONNRESET	Connection reset by peer. A peer forcibly closed the connection. This normally results from a loss of 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 queue was full.
007:021	EISCONN	Socket is already connected. The connection request was made for an already connected socket. Sending a `sendto()` call to an already connected destination could cause this error.

Table B-12.OS-9-Specific Internet Error Codes (Continued)

Number	Name	Description
007:022	ENOTCONN	Socket is not connected. A request to send or received data was rejected because the socket was not connected or no destination was given for a datagram socket.
007:023	ESHUTDOWN	Cannot send after socket shutdown. Additional data transmissions are not allowed after the socket was shut down.
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 set period of time. The time out period depends on the protocol used.
007:026	ECONNREFUSED	Connection refused by target. No connection could be established because the target machine actively refused it. This usually results from trying to connect to an inactive service 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) calls.
007:028	ESMODEXISTS	Socket module already attached. An attach was requested of an already attached socket.
007:029	ENOTSOCK	Path is not a socket. A socket function was attempted on a path that is not a socket.
007:030	EHOSTUNREACH	No route to host.
007:031	EHOSTDOWN	Host is down.
008:001	EOS_LNKDWN	Layer 1 link down (driver).
008:002	EOS_CONN	Connection error (driver).
008:003	EOS_RXTHREAD	Error in receive thread.
008:004	EOS_ME	Management entity error.
008:005	EOS_SAPI	Unrecognized SAPI
008:006	EOS_TEI	TEI not found.

Table B-12.OS-9-Specific Internet Error Codes (Continued)

Number	Name	Description
008:007	EOS_MAX_TEI	Maximum number of TEI.
008:008	EOS_TSTATE	Illegal TEI state.
008:009	EOS_TEI_DENIED	TEI request denied by network.
008:010	EOS_PRIMITIVE	TEI request denied by network.
008:011	EOS_L2IN	Layer 2 error.
008:012	EOS_PEER_BUSY	Peer receiver busy condition
008:013	EOS_K	Maximum number of outstanding I frames.
008:014	EOS_MAXCREF	Maximum number of call references.
008:015	EOS_CREF	Call reference does not exist.
008:016	EOS_CALLPROG	Error on call progress state.
008:017	EOS_RCVR	Receiver previously assigned
008:018	EOS_REQDENIED	Request denied by far end.
008:019	EOS_RXSTART	Receive thread did not start.
008:020	EOS_NOSTACK	Last driver on path’s stack.
008:021	EOS_BTMSTK	Attempt to pop last driver.
008:022	EOS_NPBNULL	Notify param block is null.
008:023	EOS_PPS_NOTFND	Per path storage not found.
008:024	EOS_STKFULL	Path’s stack array is full.
008:025	EOS_MBNOTINST	Sysmbuf is not installed.
008:026	EOS_TMRNTFND	Timer not found.
008:027	EOS_GETIME	Get time error.
008:028	EOS_TIMERINT	Timer interrupt.
008:029	EOS_RXMB_ NODEVENTRY	No device entry in mbuf.
008:030	EOS_PGM_TBLBSY	PSI/SI table is in use.
008:031	EOS_PGM_LOVF	Too many tables being read.
008:032	EOS_PGM_TBLNFND	Table not found.
008:033	EOS_PGM_NFND	Program not found.
008:034	EOS_PGM_NOPLAY	No program is currently playing.
008:035	EOS_NODNDRVR	No down driver.

Table B-13.IEEE 1394 Error Codes

Number	Name	Description
008:257	`EOS_MW1394_BUSRESET`	Bus reset. An IEEE 1394 Bus is reset whenever the cable is plugged or unplugged or a device forces a bus reset (such as when driver software is initialized).
008:258	`EOS_MW1394_NOIRM`	No Isochronous Resource Manager found.
008:259	`EOS_MW1394_NNF`	Requested node not found on the bus.
008:260	`EOS_MW1394_TIMEDOUT`	Timed out waiting for response. The remote device is not present, does not have memory mapped at the specified location, or does not allow data transfer to that location. This is a general error indicating communication was not successful.
008:261	`EOS_MW1394_FAILED`	Internal failure.
008:262	`EOS_MW1394_BADSIZE`	Bad size. The length of the packet (or response) specified is either greater than the maximum allowable payload or the allocated Asynchronous transmit buffer size.
008:263	`EOS_MW1394_ADDRINUSE`	Address in use. The address could not be mapped because some portion of the requested range has already been mapped.

Table B-13.IEEE 1394 Error Codes

Number	Name	Description
008:264	EOS_MW1394_ADDRNOTFND	Address not found. The address is not currently mapped, so it need not be unmapped.
008:265	EOS_MW1394_ADDRNOTAVAIL	Address not available. The desired address is not available for mapping.
008:266	EOS_MW1394_INVCHANNEL	Invalid channel. The channel value specified is out of range.
008:267	EOS_MW1394_CHNLINUSE	Channel in use. The specific channel requested is currently in use.
008:268	EOS_MW1394_NOCFGREG	No free Isochronous configuration registers.
008:269	EOS_MW1394_CHNLNOTFND	Channel not found. The specified channel was not found in the isochronous configuration registers.
008:270	EOS_MW1394_CHNL_STOPPED	Channel is stopped. Operations on this channel have been stopped by a call to ms1394IsochStop().
008:271	EOS_MW1394_NO_FREECHNL	Channel deallocation failed.
008:272	EOS_MW1394_NOXMIT	Could not transmit. Data may not be transmitted because the FIFO is full or a hardware condition is prohibiting transmit.
008:273	EOS_MW1394_INVTOPOMAP	Topology Map invalid. The Topology Map on the Bus Manager is invalid.
008:274	EOS_MW1394_INVSPEEDMAP	Invalid speedmap.
008:275	EOS_MW1394_BUSMGR_EXISTS	Bus Manager already present. There is already a Bus Manager present.

Table B-13.IEEE 1394 Error Codes

Number	Name	Description
008:276	EOS_MW1394_BEBUSMGRFAIL	IRM refused request. The request to be a bus manager failed because the IRM (Isochronous Resource Manager) did not honor the request.
008:277	EOS_MW1394_BUSMGR_ALRDY	Already the Bus Manager. The requesting node is already the Bus Manager.
008:278	EOS_MW1394_ALLOCCHNLFAIL	Channel allocation failed.
008:279	EOS_MW1394_FREECHNLFAIL	Channel deallocation failed.
008:280	EOS_MW1394_CHNLFREEALRDY	Channel already free. The specified channel is already free.
008:281	EOS_MW1394_ALLOCBWDTHFAIL	Failed to allocate bandwidth.
008:282	EOS_MW1394_FREEBWDTHFAIL	Failed to deallocate bandwidth.
008:283	EOS_MW1394_BANDWIDTH_ NOTAVAIL	Bandwidth is not available. The amount of bandwidth requested is not available. The caller may reduce the desired maxbytesPerFrame and try again or retry after some delay.
008:284	EOS_MW1394_INV_BANDWIDTH_ HNDL	Invalid handle. No bandwidth handle is found with the specified handleID.
008:285	EOS_MW1394_NOBUFFERS_ ATTCHD	No buffers attached. No buffers are attached for the specified channel.

Table B-13.IEEE 1394 Error Codes

Number	Name	Description
008:286	EOS_MW1394_RCODE_ERROR	Remote response code was not RCODE_COMPLETE. A response code other than RCODE_COMPLETE was received.
008:287	EOS_MW1394_NOBUSMGR	No Bus Manager found.
008:288	EOS_MW1394_INVACCESSTYPE	Invalid access type. No valid access type was specified.

Table B-14.POSIX Error Codes

Number	Name	Description
020:001	E2BIG	Argument list too long.
020:003	EBADMSG	Bad message.
020:004	ECANCELED	Operation canceled.
020:005	EDEADLK	Resource deadlock avoided.
020:006	EFBIG	File too big.
020:007	EISDIR	Is a directory.
020:008	EMFILE	Too many process open files.
020:009	EMLINK	Too many links.
020:010	ENAMETOOLONG	Filename too long.
020:011	ENFILE	Too many system open files.
020:012	ENOEXEC	Exec format error.
020:013	ENOLCK	No locks available.
020:014	ENOSPC	No space left on device.
020:015	ENOSYS	Function not implemented.
020:016	ENOTDIR	Not a directory.
020:017	ENOTEMPTY	Directory not empty.
020:018	ENOTSUP	Not supported.
020:019	ENOTTY	Bad I/O control operation.
020:020	EROFS	Read-only file system.

Table B-14.POSIX Error Codes

Number	Name	Description
020:021	ESPIPE	Invalid seek.
020:022	EXDEV	Improper link.

Table B-15.OS-9-Specific PowerPC Error Codes

Number	Name	Description
100:002	EOS_PPC_MACHCHK	Machine check exception.
100:003	EOS_PPC_DATAACC	Data access exception.
100:004	EOS_PPC_INSTACC	Instruction access exception.
100:005	EOS_PPC_EXTINT	External interrupt.
100:006	EOS_PPC_ALIGN	Alignment exception.
100:007	EOS_PPC_PROGRAM	Program exception.
100:008	EOS_PPC_FPUUNAV	FPU unavailable exception.
100:009	EOS_PPC_DEC	Decrementer exception.
100:010	EOS_PPC_IOCONT	I/O controller exception.
100:012	EOS_PPC_SYSCALL	System call exception.
100:032	EOS_PPC_TRACE	Trace exception.

Table B-16.OS-9-Specific MIPS Error Codes

Number	Name	Description
102:000	EOS_MIPS_EXTINT	External interrupt.
102:001	EOS_MIPS_MOD	TLB Modification exception.
102:002	EOS_MIPS_TLBL	TLB Miss exception (load or instruction fetch).
102:003	EOS_MIPS_TLBS	TLB Miss exception (store).
102:004	EOS_MIPS_ADEL	Address Error exception (load or instruction fetch).
102:005	EOS_MIPS_ADES	Address Error exception (store).
102:006	EOS_MIPS_IBE	Bus Error exception (instruction fetch).

Table B-16.OS-9-Specific MIPS Error Codes

Number	Name	Description
102:007	EOS_MIPS_DBE	Bus Error exception (load or store).
102:008	EOS_MIPS_SYS	SYSCALL exception.
102:009	EOS_MIPS_BP	Breakpoint exception.
102:010	EOS_MIPS_RI	Reserved Instruction exception.
102:011	EOS_MIPS_CPU	CoProcessor Unusable exception.
102:012	EOS_MIPS_OVF	Arithmetic Overflow exception.
102:013	EOS_MIPS_TR	Trap exception.
102:023	EOS_MIPS_WATCH	Watch exception.
102:032	EOS_MIPS_UTLB	User State TLB Miss exception.

Table B-17.OS-9-Specific ARM Error Codes

Number	Name	Description
103:001	EOS_ARM_UNDEF	Undefined instruction exception.
103:003	EOS_ARM_PFABORT	Instruction pre-fetch abort exception.
103:004	EOS_ARM_DTABORT	Data abort exception.
103:008	EOS_ARM_ALIGN	Alignment exception.

Index

S A B C D E F G H I J K L M N O P Q R S T U V W X Y Z