OS-9 Operating System

System Programmer's Manual

OS-9 Operating System: System Programmer's Manual

Revision History

Revision F-1 January 1983
Original Microware edition
Revision G January 2003
Updated to reflect OS-9 Level 1 Version 1.2

Table of Contents

Chapter 1. Introduction

OS-9 Level One is a versatile multiprogramming/multitasking operating system for computers utilizing the Motorola 6809 microprocessor,. It is well-suited for a wide range of applications on 6809 computers of almost any size or complexity. Its main features are:

- Comprehensive management of all system resources: memory, input/output and CPU time.
- A powerful user interface that is easy to learn and use.
- True multiprogramming operation.
- Efficient operation in typical microcomputer configuratjons.
- Expandable, device-independent unified I/O system.
- Full support for modular ROMed software.
- Upward and downward compatibility with OS-9 Level Two.

This manual is intended to provide the information necessary to install, maintain, expand, or write assembly-language software for OS-9 systems. It assumes that the reader is familiar with the 6809 architecture, instruction set, and assembly language.

History And Design Philosophy

OS-9 Level One is one of the products of the BASIC09 Advanced 6809 Programming Language development effort undertaken by Microware and Motorola from 1978 to 1980. During the course of the project it became evident that a fairly sophisticated operating system would be required to support BASIC09 and similar high-performance 6809 software.

OS-9's design was modeled after Bell Telephone Laboratories' UNIX® operating system, which is becoming widely recognized as a standard for mini and micro multiprogramming operating systems because of its versatility and relatively simple, yet elegant structure. Even though a "clone" of UNIX for the 6809 is relatively easy to implement, there are a number of problems with this approach. UNIX was designed for fairly large-scale minicomputers (such as large PDP-11s) that have high CPU throughput, large fast disk storage devices and a static I/O environment. Also, UNIX is not particularly time or disk-storage efficient, especially when used with low-cost disk drives.

For these reasons, OS-9 was designed to retain the overall concept and user interface of UNIX, but its implementation is considerably different. OS-9's design is tailored to typical microcomputer performance ranges and operational environments. As an example, OS-9, unlike UNIX, does not dynamically swap running programs on and off disk This is because floppy disks and many lower-cost Winchester-type hard disks are simply too slow to do this efficiently. Instead, OS-9 always keeps running programs in memory and emphasizes more efficient use of available ROM or RAM.

OS-9 also introduces some important new features that are intended to make the most of the capabilities of third-generation microprocessors, such as support of reentrant, position-independent software that can be shared by several users simultaneously to reduce overall memory requirements.

Perhaps the most innovative part of OS-9 is its "memory module" management system, which provides extensive support for modular software, particularly ROMed software. This will play an increasingly important role in the future as a method of reducing software costs. The "memory module" and LINK capabilities of OS-9 permit modules to be automatically identified, linked together, shared, updated or repaired. Individual modules in ROM which are defective may be repaired (without reprogramming the ROM) by placing a "fixed" module, with the same name,

but a higher revision number into memory. Memory modules have many other advantages, for example, OS-9 can allow several programs to share a common math subroutine module. The same module could automatically be replaced with a module containing drivers for a hardware arithmetic processor without any change to the programs which call the module.

Users experienced with UNIX should have little difficulty adapting to OS-9. Here are some of the main differences between the two systems:

1. OS-9 is written in 6809 assembly language, not C. This improves program size and speed characteristics.
2. OS-9 was designed for a mixed RAM/ROM microcomputer memory environment and more effectively supports reentrant, position-independent code.
3. OS-9 introduces the "memory module" concept for organizing object code with built-in dynamic inter-module linkage.
4. OS-9 supports multiple file managers, which are modules that interface a class of devices to the file system.
5. "Fork" and "Execute" calls are faster and more memory efficient than the UNIX equivalents.

System Hardware Requirements

The OS-9 Operating system consists of building blocks called memory modules, which are automatically located and linked together when the system starts up. This makes it extremely easy to reconfigure the system. For example, reconfiguring the system to handle additional devices is simply a matter of placing the corresponding modules into memory. Because OS-9 is so flexible, the minimum hardware requirements are difficult to define. A bare-bones LEVEL I system requires 4K of ROM and 2K of RAM, which may be expanded to 56K RAM.

Shown below are the requirements for a typical OS-9 software development system. Actual hardware requirements may vary depending upon the particular application.

- 6809 MPU
- 24K Bytes RAM Memory for Assembly Language Development. 40K Bytes RAM Memory for High Level Languages such as BASIC09 (RAM Must Be Contiguous From Address Zero Upward)
- 4K Bytes of ROM: 2K must be addressed at $F800 - $FFFF, the other 2K is position-independent and self-locating. Some disk systems may require three 2K ROMs.
- Console terminal and interface using serial, parallel, or memory mapped video.
- Optional printer using serial or parallel interface.
- Optional real-time clock hardware.

I/O device controller addresses can be located anywhere in the memory space, however it is good practice to place them as high as possible to maximize RAM expansion capability. Standard OS-9 packages for computers made by popular manufacturers usually conform to the system's customary memory map.

Chapter 2. Basic System Organization

OS-9 is composed of a group of modules, each of which provides specific functions. When OS-9 is configured for a specific system various modules are selected to provide a given level of functionality. For example, a small control computer without a disk does not need the disk-related OS-9 modules. Most examples in this manual describe a fully-configured OS-9 system.

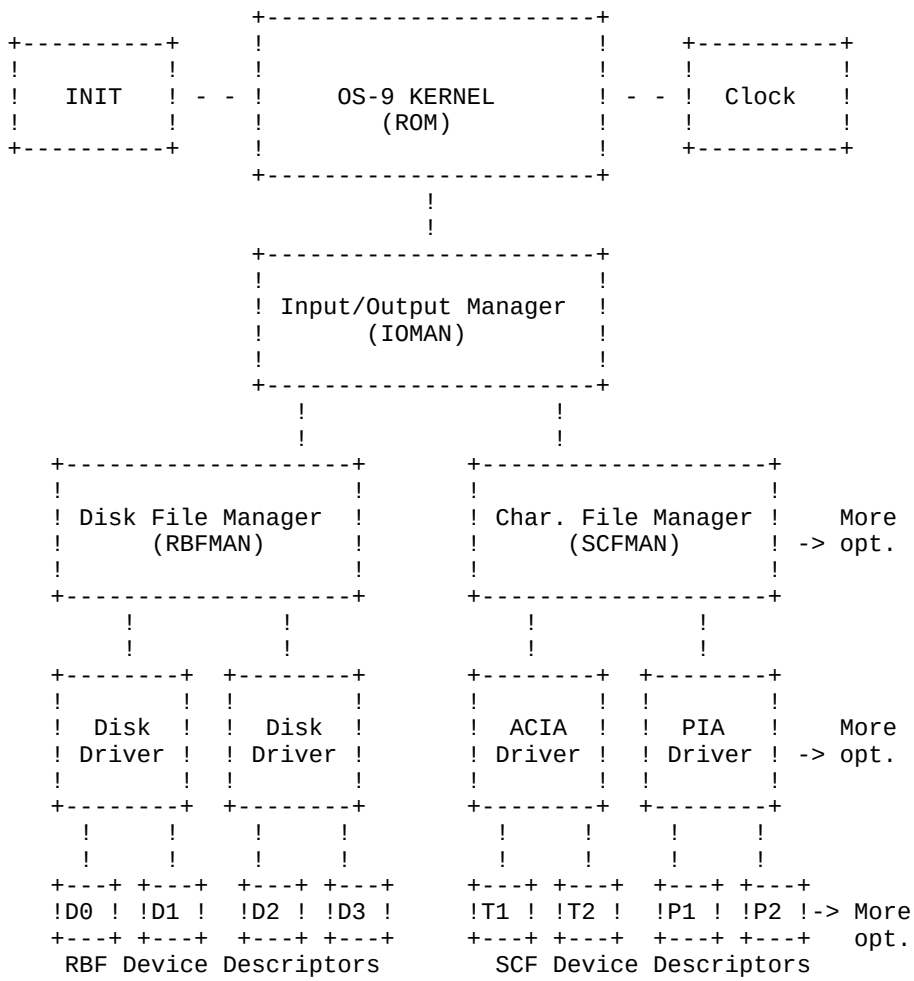


Figure 2-1. OS-9 Component Module Organization

Notice that the diagram on the previous page indicates a multilevel organization.

The first level is the KERNEL and the CLOCK MODULE. The kernel provide basic system services such as multitasking, memory management, and links all other system modules. The CLOCK module is a software handler for the specific real-time-clock hardware. INIT is an initialization table used by the kernel during system startup. It specifies initial table sizes, initial system device names, etc.

The second level is the Input/Output Manager. It provides common processing all I/O operations. It is required if any OS-supported I/O is to be performed.

The third level is the File Manager level. File managers perform I/O request processing for similar classes of I/O devices. The Random Block File Manager (RBFMAN) processes all disk-type device functions, and the Sequential Character File Manager (SCFMAN) handles all non-mass storage devices that basically operate a character at a time, such as terminals and printers. The user can add additional File Managers to handle classes of devices not covered by SCFMAN or RBFMAN.

The fourth level is the Device Driver Level. Device drivers handle basic physical I/O functions for specific I/O controller hardware. Standard OS-9 systems are typically supplied with a disk driver, a ACIA driver for terminals and serial printers, and a PIA driver for parallel printers. Many users add customized drivers of their own design or purchased from a hardware vendor.

The fifth level is the Device Descriptor Level. These modules are small tables that are associate specific I/O ports with their logical names, and the port's device driver and file manager. They also contain the physical address of the port and initialization data. By use of device descriptors, only one copy of each driver is required for each specific type of I/O controller regardless of how many controllers the system uses.

One important component not shown is the `shell`, which is the command interpreter. It is technically a program and not part of the operating system itself, and is described fully in the OS-9 Users Manual.

Even though all modules can be resident in ROM, generally only the KERNEL and INIT modules are ROMed in disk-based systems. All other modules are loaded into RAM during system startup by a disk bootstrap module (not shown on diagram) which is also resident in ROM.

Chapter 3. Basic Functions of the Kernel

The nucleus of OS-9 is the "kernel", which serves as the system administrator, supervisor, and resource manager. It is about 3K bytes long and normally resides in two 2K byte ROMs: "P1" residing at addresses $F800 - $FFFF, and "P2", which is position-independent. P2 only occupies about half (1K) of the ROM, the other space in the ROM is reserved for the disk bootstrap module.

The kernel's main functions are:

1. System initialization after restart.
2. Service request processing.
3. Memory management.
4. MPU management (multiprogramming).
5. Basic interrupt processing.

Notice that input/output functions were not included in the list above; this is because the kernel does not directly process them. The kernel passes I/O service requests directly to another the Input/Output Manager (IOMAN) module for processing.

After a hardware reset, the kernel will initialize the system which involves: locating ROMs in memory, determining the amount of RAM available, loading any required modules not already in ROM from the bootstrap device, and running the system startup task ("SYSGO"). The INIT module is a table used during startup to specify initial table sizes and system device names.

Kernel Service Request Processing

Service requests (system calls) are used to communicate between OS-9 and assembly-language-level programs for such things as allocating memory, creating new processes, etc. System calls use the SWI2 instruction followed by a constant byte representing the code. Parameters for system calls are usually passed in MPU registers. In addition to I/O and memory management functions, there are other service request functions including interprocess control and timekeeping.

A system-wide assembly languaqe equate file called `OS9Defs` defines symbolic names for all service requests. This file is included when assembling hand-written or compiler-generated code. The OS-9 Assembler has a built-in macro to generate system calls, for example:

```
OS9 I$Read
```

is recongnized and assembled as the equivalent to:

```
SWI2
FCB  I$Read
```

Service requests are divided into two categories:

I/O REQUESTS perform various input/output functions. Requests of this type are passed by the kernel to IOMAN for processing. The symbolic names for this category have a "I$" prefix, for example, the "read" service request is called I$Read.

FUNCTION REQUESTS perform memory management, multiprogramming, and miscellaneous functions. Most are processed by the kernel. The symbolic names for this category begins with "F$".

Kernel Memory Management Functions

Memory management is an important operating system function. OS-9 manages both the physical assignment of memory to programs *and* the logical contents of memory, by using entities called "memory modules". All programs are loaded in memory module format, allowing OS-9 to maintain a directory which contains the name, address, and other related information about each module in memory. These structures are the foundation of OS-9's modular software environment. Some of its advantages are: automatic run-time "linking" of programs to libraries of utility modules; automatic "sharing" of reentrant programs; replacement of small sections of large programs for update or correction (even when in ROM); etc.

Memory Utilization

All usable RAM memory must be contiguous from address 0 upward. During the OS-9 start-up sequence the upper bound of RAM is detemined by an automatic search, or from the configuration module. Some RAM is reserved by OS-9 for its own data structures at the top and bottom of memory. The exact amount depends on the sizes of system tables that are specified in the configuration module.

All other RAM memory is pooled into a "free memory" space. Memory space is dynamically taken from and returned to this pool as it is allocated or deallocated for various purposes. The basic unit of memory allocation is the 256-byte page . Memory is always allocated in whole numbers of pages.

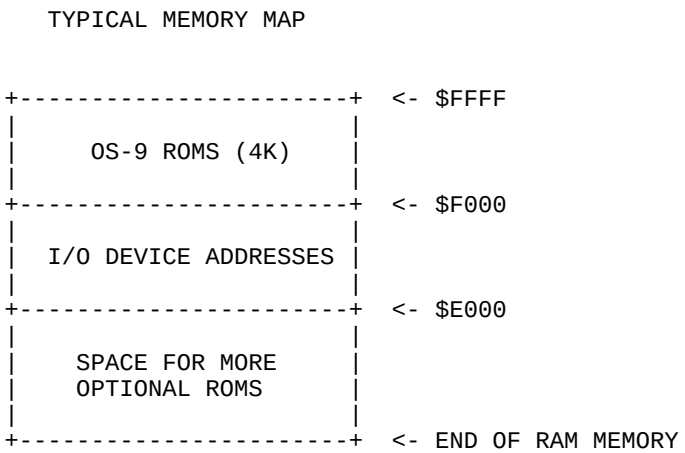
The data structure used to keep track of memory allocation is a 32-byte bit-map located at addresses $0100 - $011F. Each bit in this table is associated with a specific page of memory. Bits are cleared to indicate that the page is free and available for assignment, or set to indicate that the page is in use or that no RAM memory is present at that address.

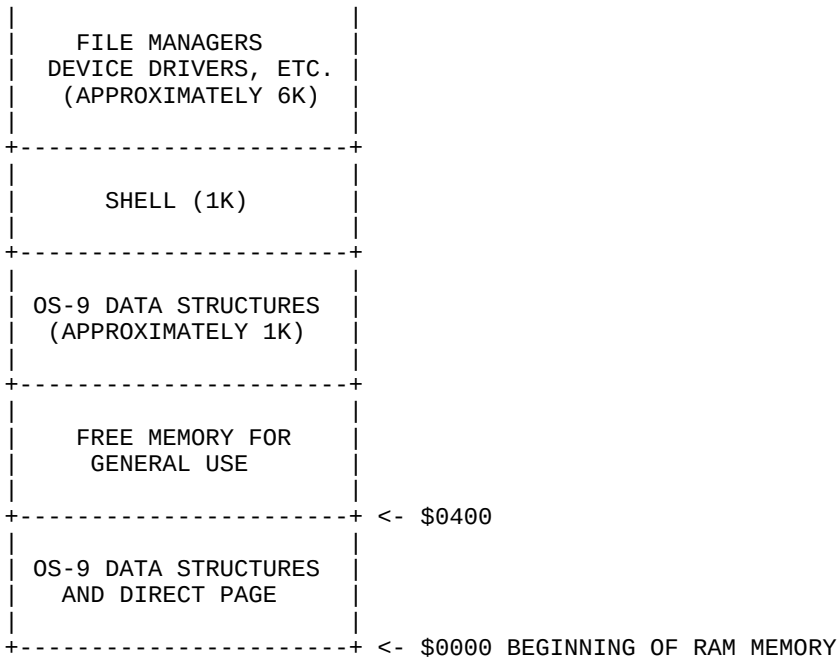
Automatic memory allocation occurs when:

1. Program modules are loaded into RAM.
2. Processes are created.
3. Processes request additional RAM.
4. OS-9 needs I/O buffers, larger tables, etc.

All of the above usually have inverse functions that cause previously allocated memory to be deallocated and returned to the free memory pool.

In general, memory is allocated for program modules and buffers from high addresses downward, and for process data areas from lower addresses upward.





The map above is for a “typical” system. Actual memory sizes and addresses may vary depending on the exact system configuration.

Overview of Multiprogramming

OS-9 is a multiprogramming operating system, which allows several independent programs called “processes” can be executed simultaneously. Each process can have access to any system resource by issuing appropriate service requests to OS-9. Multiprogramming functions use a hardware real-time clock that generates interrupts at a regular rate of about 10 times per second. MPU time is therefore divided into periods typically 100 milliseconds in duration. This basic time unit is called a tick . Processes that are “active” (meaning not waiting for some event) are run for a specific system-assigned period called a “time slice”. The duration of the time slice depends on a process’s priority value relative to the priority of all other active processes. Many OS-9 service requests are available to create, terminate, and control processes.

Process Creation

New processes are created when an existing process executes a F$Fork service request. Its main argument is the name of the program module (called the “primary module”) that the new process is to initially execute. OS-9 first attempts to find the module in the “module directory”, which includes the names of all program modules already present in memory. If the module cannot be found there. OS-9 usually attempts to load into memory a mass-storage file using the requested module name as a file name.

Once the module has been located, a data structure called a “process descriptor” is assigned to the new process. The process descriptor is a 64-byte package that contains information about the process, its state, memory allocations, priority, queue pointers, etc. The process descriptor is automatically initialized and maintained by OS-9. The process itself has no need, and is not permitted to access the descriptor.

The next step in the creation of a new process is allocation of data storage (RAM) memory for the process. The primary module’s header contains a storage size value that is used unless the F$Fork system call requested an optionally larger size. OS-9

then attempts to allocate a CONTIGUOUS memory area of this size from the free memory space.

If any of the previous steps cannot be performed, creation of the new process is aborted, and the process that originated the F$Fork is informed of the error. Otherwise, the new process is added to the active process queue for execution scheduling.

The new process is also assigned a unique number called a "process ID" which is used as its identifier. Other processes can commnunciate with it by referring to its ID in various system calls. The process also has associated with it a "user ID" which is used to identify all processes and files belonging to a particular user. The user ID is inherited from the parent process.

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

Process States

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

ACTIVE - The process is active and ready for execution.

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

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

There is a queue for each process state. The queue is a linked list of the "process descriptors" of processes in the corresponding state. State changes are performed by moving a process descriptor to another queue.

The Active State

This state includes all "runnable" processes, which are given time slices for execution according to their relative priority with respect to all other active processes. The scheduler uses a pseudo-round-robin scheme that gives all active processes some CPU time, even if they have a very low relative priority.

The Wait State

This state is entered when a process executes a F$Wait system service request. The process remains suspended until the death of any of its descendant processes, or, until it receives a signal.

The Sleeping State

This state is entered when a process executes a F$Sleep service request, which specifies a time interval. (a specific number of ticks) for which the process is to remain suspended. The process remains asleep until the specified time has elapsed, or until a signal is received.

Execution Scheduling

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

All active processes are members of the active process queue, which is kept sorted by process "age". Age is a count of how many process switches have occurred since the process' last time slice. When a process is moved to the active process queue from another queue, its "age" is initialized by setting it to the process' assigned priority, i.e., processes having relatively higher priority are placed in the queue with an artificially higher age. Also, whenever a new process is activated, the ages of all other processes are incremented.

Upon conclusion of the currently executing process' time slice, the scheduler selects the process having the highest age to be executed next. Because the queue is kept sorted by age, this process will be at the bead of the queue. At this time the ages of all other active processes are incremented (ages are never incremented beyond 255).

An exception is newly-active processes that were previously deactivated while they were in the system state. These processes are noted and given higher priority than others because they are usually executing critical routines that affect shared system resources and therefore could be blocking other unrelated processes.

When there are no active processes, the kernel will set itself up to handle the next interrupt and then execute a CWAI instruction, which decreases interrupt latency time.

Signals

"Signals" are an asynchronous control mechanism used for interprocess communication and control. A signal behaves like a software interrupt in that it can cause a process to suspend a program, execute a specific routine, and afterward return to the interrupted program. Signals can be sent from one process to another process (by means of the SEND service request), or they can be sent from OS-9 system routines to a process.

Status information can be conveyed by the signal in the form of a one-byte numeric value. Some of the signal "codes" (values) have predefined meanings, but all the rest are user-defined. The defined signal codes are:

0 = KILL (non-interceptable process abort)

1 = WAKEUP - wake up sleeping process

2 = KEYBOARD ABORT

3 = KEYBOARD INTERRUPT

4 - 255 USER DEFINED

When a signal is sent to a process, the signal is noted and saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. It then becomes eligible for execution according to the usual MPU scheduler criteria. When it gets its next time slice, the signal is processed.

What happens next depends on whether or not the process had previously set up a "signal trap" (signal service routine) by executing an F$ICPT service request. If it had not, the process is immediately aborted. It is also aborted if the signal code is zero. The abort will be deferred if the process is in system mode: the process dies upon its return to user state.

If a signal intercept trap has been set up, the process resumes execution at the address given in the F$ICPT service request. The signal code is passed to this routine, which should terminate with an RTI instruction to resume normal execution of the process.

NOTE: "Wakeup" signals activate a sleeping process: they *do not* vector through the intercept routine.

If a process has a signal pending (usually because it has not been assigned a time slice since the signal was received), and some other process attempts to send it another signal, the new signal is aborted and the "send" service request will return an error status. The sender should then execute a sleep service request for a few ticks before attempting to resend the signal, so the destination process has an opportunity to process the previously pending signal.

Interrupt Processing

Interrupt processing is another important function of the kernel. All hardware interrupts are vectored to specific processing routines. IRQ interrupts are handled by a prioritized polling system (actually part of IOMAN) which automatically identifies the source of the interrupt and dispatches to the associated user or system defined service routine. The real-time clock will generate IRQ interrupts. SWI, SWI2, and SWI3 interrupts are vectored to user-definable addresses which are "local" to each procedure, except that SWI2 is normally used for OS-9 service requests calls. The NMI and FIRQ interrupts are not normally used and are vectored through a RAM address to an RTI instruction.

Physical Interrupt Processing

The OS-9 kernel. ROMs contain the hardware vectors required by the 6809 MPU at addresses $FFF0 through $FFFF. These vectors each point to jump-extended-indirect instruction which vector the MPU to the actual interrupt service routine. A RAM vector table in page zero of memory contains the target addresses of the jump instructions as follows:

INTERRUPT	ADDRESS
SWI3	$002C
SWI2	$002E
FIRQ	$0030
IRQ	$0032
SWI	$0034
NMI	$0036

OS-9 initializes each of these locations after reset to point to a specific service routine in the kernel. The SWI, SWI2, and SWI3 vectors point to specific routines which in turn read the corresponding pseudo vector from the process' process descriptor and dispatch to it. This is why the F$SSWI service request to be local to a process since it only changes a pseudo vector in the process descriptor. The IRQ routine points directly to the IRQ polling system, or to it indirectly via the real-time clock device service routine. The FIRQ and NMI vectors are not normally used by OS-9 and point to RTI instructions.

A secondary vector table located at $FFE0 contains the addresses of the routines that the RAM vectors are initialized to. They may be used when it is necessary to restore the original service routines after altering the RAM vectors. On the next page are the definitions of both the actual hardware interrupt vector table, and the secondary vector table:

VECTOR	ADDRESS	
Secondary Vector Table		
TICK	$FFE0	Clock Tick Service Routine
SWI3	$FFE2	

VECTOR	ADDRESS	
SWI2	$FFE4	
FIRQ	$FFE6	
IRQ	$FFE8	
SWI	$FFEA	
NMI	$FFEC	
WARM	$FFEE	Reserved for warm-start
Hardware Vector Table		
SWI3	$FFF2	
SWI2	$FFF4	
FIRQ	$FFF6	
IRQ	$FFF8	
SWI	$FFFA	
NMI	$FFFC	
RESTART	$FFFE	

If it is necessary to alter the RAM vectors use the secondary vector table to exit the substitute routine. The technique of altering the IRQ pointer is usually used by the clock service routines to reduce latency time of this frequent interrupt source.

Logical Interrupt Polling System

In OS-9 systems, most I/O devices use IRQ-type interrupts, so OS-9 includes a sophisticated polling system that automatically identifies the source of the interrupt and dispatches to its associated user-defined service routine. The information required for IRQ polling is maintained in a data structure called the "IRQ polling table". The table has a 9-byte entry for each possible IRQ-generatinq device. The table size is static and defined by an initialization constant in the System Configuration Module.

The polling system is prioritized so devices having a relatively greater importance (i.e., interrupt frequency) are polled before those of lesser priority. This is accomplished by keeping the entries sorted by priority, which is a number between 0 (lowest) and 255 (highest). Each entry in the table has 6 variables:

1. POLLING ADDRESS: The address of the device's status register, which must have a bit or bits that indicate it is the source of an interrupt.
2. MASK BYTE; This byte selects one or more bits within the device status register that are interrupt request flag(s). A set bit identifies the active bit(s).
3. FLIP BYTE: This byte selects whether the bits in the device status register are true when set or true when cleared. Cleared bits indicate active when set.
4. SERVICE ROUTINE ADDRESS: The user-supplied address of the device's interrupt service routine.
5. STATIC STORAGE ADDRESS: a user-supplied ter to the permanent storage required by the device service routine.
6. PRIORITY; The device priority number: 0 to 255. This value determines the order in which the devices in the polling table will be polled. Note: this is not the same as a process priority which is used by the execution scheduler to decide which process gets the next time slice for MPU execution.

When an IRQ interrupt occurs, the polling system is entered via the corresponding RAM interrupt vector. It starts polling the devices, using the entries in the polling

table in priority order. For each entry, the status register address is loaded into accumulator A using the device address from the table. An exclusive-or operation using the flip-byte is executed, followed by a logical-and operation using the mask byte. If the result is non-zero, the device is assumed to be the cause of the interrupt.

The device's static storage address and service routine address is read from the table and executed.

> **Note:** The interrupt service routine should terminate with an an *RTS*, not an RTI instruction.

Entries can be made to the IRQ polling table by use of a special OS-9 service request called F$IRQ. This is a priviledged service request that can be executed only when OS-9 is in System Mode (which is the case when device drivers are executed).

> **Note:** The actual code for the interrupt polling system is located in the IOMAN module. The kernel P1 and P2 modules contain the physical interrupt processing routines.

Chapter 4. Memory Modules

Any object to be loaded into the memory of an OS-9 system must use the memory module format and conventions. The memory module concept allows OS-9 to manage the logical contents as well as the physical contents of memory. The basic idea is that all programs are individual, named objects.

The operating system keeps track of modules which are in memory at all times by use of a "module directory" . It contains the addresses and a count of how many processes are using each module. When modules are loaded into memory, they are added to the directory. When they are no longer needed, their memory is deallocated and their name removed from the directory (except ROMs, which are discussed later). In many respects, modules and memory in general, are managed just like a disk. In fact, the disk and memory management sections of OS-9 share many subroutines.

Each module has three parts; a module header, module body and a cyclic-redundancy-check (CRC) value. The header contains information that describes the module and its use. This information includes: the modules size, its type (machine language. BASIC09 compiled code, etc); attributes (executable, reentrant, etc), data storage memory requirements, execution starting address, etc. The CRC value is used to verify the integrity of a module.

There are several different kinds of modules, each type having a different usage and function. Modules do not have to be complete programs, or even 6809 machine language. They may contain BASIC09 "I-code", constants, single subroutines, subroutine packages, etc. The main requirements are that modules do not modify themselves arid that they be position-independent so OS-9 can load or relocate them wherever memory space is available. In this respect, the module format is the OS-9 equivalent of "load records" used in older-style operating systems.

Memory Module Structure

At the beginning (lowest address) of the module is the module header, which can have several forms depending on the module's usage. OS-9 family software such as BASIC09, Pascal, C, the assembler, and many utility programs automatically generate modules and headers. Following the header is the program/constant section which is usually pure code. The module name string is included somewhere in this area. The last three bytes of the module are a three-byte Cyclic Redundancy Check (CRC) value used to verify the integrity of the module.

Table 4-1. Module Format

MODULE HEADER
PROGRAM OR CONSTANTS
CRC

The 24-bit CRC is performed over the entire module from the first byte of the module header to the byte just before the CRC itself. The CRC polynomial used is $800063. (See F$CRC)

Because most OS-9 family software (such as the assembler) automatically generate the module header and CRC values, the programner usually does not have to be concerned with writing routines to generate them.

Module Header Definitions

The first nine bytes of all module headers are identical:

MODULE OFFSET	DESCRIPTION
$0,$1 =	Sync Bytes ($87,$CD). These two constant bytes are used to locate modules.
$2,$3 =	Module Size. The overall size of the module in bytes (includes CRC).
$4,$5 =	Offset to Module Name. The address of the module name string relative to the start (first sync byte) of the module. The name string can be located anywhere in the module and consists of a string of ASCII characters having the sign bit set on the last character.
$6 =	Module Type/Language Type. See text.
$7 =	Attributes/Revision Level. See text.
$8 =	Header Check. The one's compliment of (the vertical parity (exclusive OR) of) the previous eight bytes

Type/Language Byte

The module type is coded into the tour most significant bits of byte 6 of the module header. Eight types are pre-defined by convention, some of which are for OS-9's internal use only. The type codes are:

$1	Program module
$2	Subroutine module
$3	Multi-module (for future use)
$4	Data module
$5-$B	User-definable
$C	OS-9 System module
$D	OS-9 File Manager module
$E	OS-9 Device Driver module
$F	OS-9 Device Descriptor module

NOTE: 0 is not a legal type code.

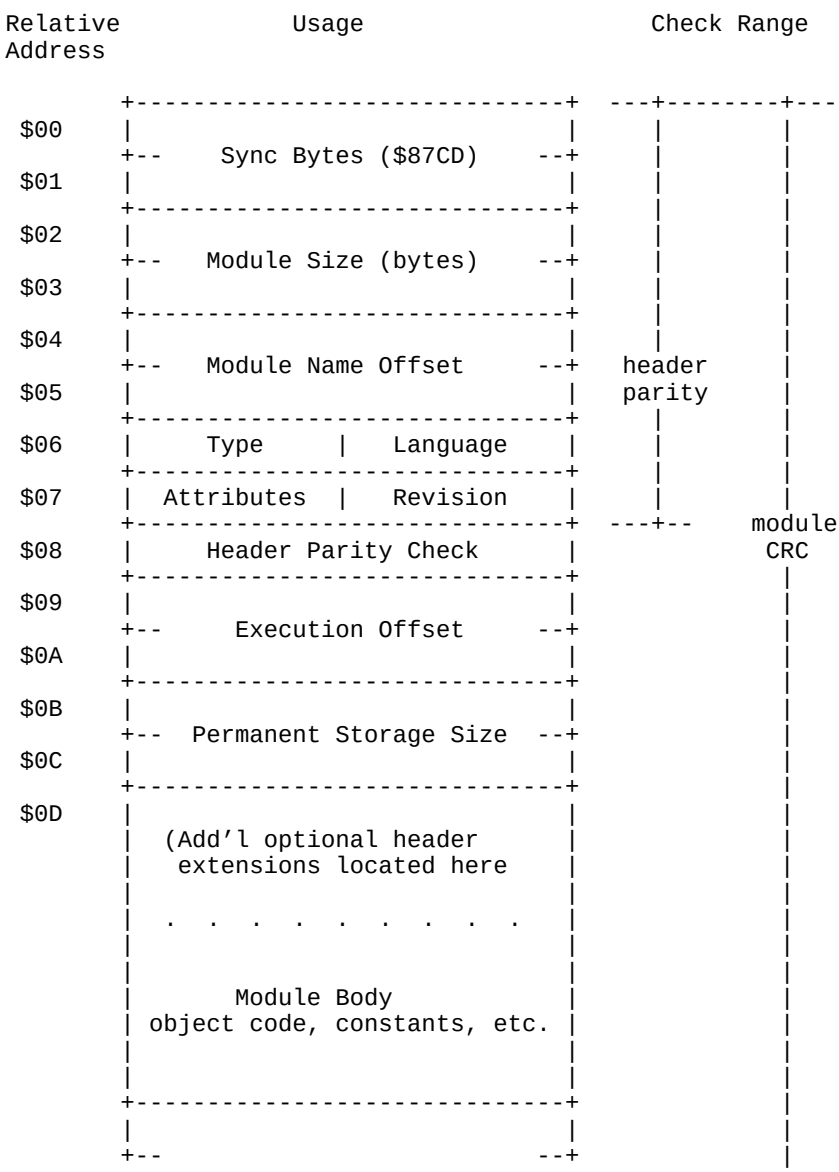
"user-defined" types having type codes of 0 through 9. They have six more bytes in their headers defined as follows:

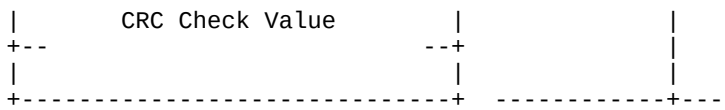
MODULE OFFSET	DESCRIPTION
$9,$A =	Execution Offset. The program or subroutine's starting address, relative to the first byte of the sync code. Modules having multiple entry points (cold start, warm start, etc.) may have a branch table starting at this address.

MODULE DESCRIPTION
OFFSET

$B,$C = Permanent Storage Requirement. This is the minimum number of bytes of data storage required to run. This is the number used by F$Fork and F$Chain to allocate a process' data area. If the module will not be directly executed by a F$Chain or F$Fork service request (for instance a subroutine package), this entry is not used by OS-9. It is commonly used to specify the maximum stack size required by reentrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

Executable Memory Module Format





ROMed Memory Modules

When OS-9 starts after a system reset, it searches the entire memory space for ROMed modules. It detects them by looking for the module header sync code ($87,$CD) which are unused 6809 opcodes. When this byte pattern is detected, the header check is performed to verify a correct header. If this test succeeds, the module size is obtained from the header and a 24-bit CRC is performed over the entire module. If the CRC matches correctly, the module is considered valid, and it is entered into the module directory. The chances of detecting a "false module" are virtually nil.

In this manner all ROMed modules present in the system at startup are automatically included in the system module directory. Some of the modules found initially are various parts of OS-9: file managers, device driver, the configuration module, etc.

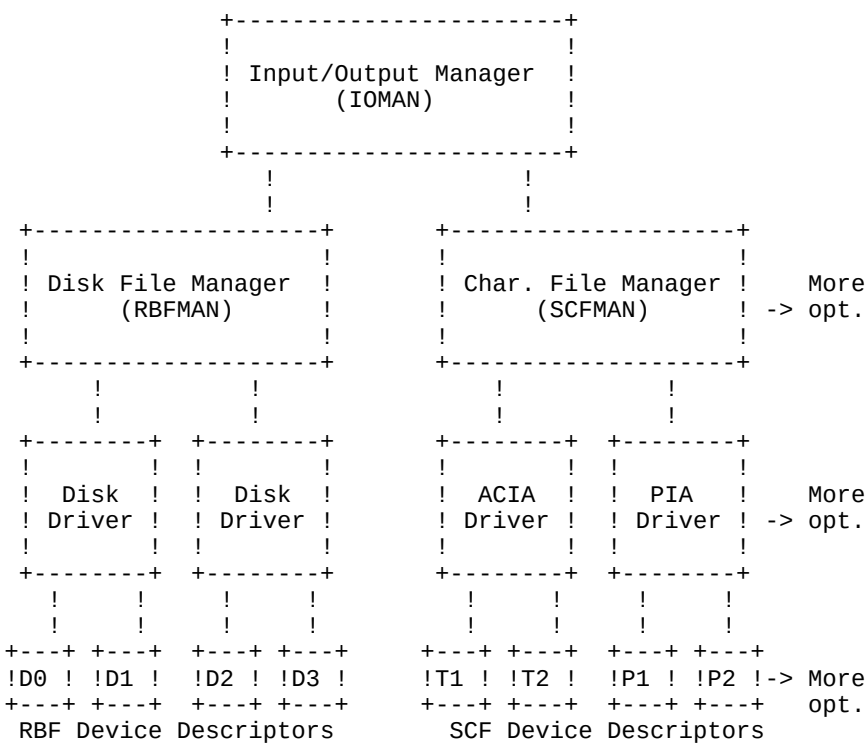
After the module search OS-9 links to whichever of its component modules that it found. This is the secret of OS-9's extraordinary adaptability to almost any 6809 computer; it automatically locates its required and optional component modules, wherever they are, and rebuilds the system each time that it is started.

ROMs containing non-system modules are also searched so any user-supplied software is located during the start-up process and entered into the module directory.

Chapter 5. The OS-9 Unified Input/Output System

OS-9 has a unified I/O system that provides system-wide hardware-independent I/O services for user programs and OS-9 itself. All I/O service requests (system call) are received by the kernel and passed to the Input/Output Manager (IOMAN) module for processing IOMAN performs some processing (such as allocating data structures for the I/O path) and calls the file managers and device drivers to do much of the actual work. File manager, device driver, and device descriptor modules are standard memory modules that can be loaded into memory from files and used while the system is running.

The structural organization of I/O-related modules in an OS-9 system is hierarchical, as illustrated below:



The Input/Output Manager (IOMAN)

The Input/output Manager (IOMAN) module provides the first level of service for I/O system calls by routing data on I/O paths from processes to/from the appropriate file managers and device drivers. It maintains two important internal OS-9 data structures: the device table and the path table. This module is used in all OS-9 Level One systems and should never be modified.

When a path is opened, IOMAN attempts to link to a memory module having the device name given (or implied) in the pathlist. This module is the device's descriptor, which contains the names of the device driver and file manager for the device. This information is saved by IOMAN so subsequent system calls can be routed to these modules.

File Managers

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

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

The file managers are reentrant and one file manager may be used for an entire class of devices having similar operational ctiaracteristics. The two standard OS-9 file managers are:

RBFMAN: The Random Block File Manager which operates random-access, block-structured devices such as disk systems, bubble memories, etc.

SCFMAN: Sequential Character File Manager which is used with single-character-oriented devices such as CRT or hardcopy terminals, printers, modems etc.

Device Driver Modules

The device driver modules are subroutine packages that perform basic, low-level I/O transfers to or from a specific type of I/O device hardware controller. These modules are reentrant so one copy of the module can simultaneously run several different devices which use identical I/O ccntrollers. For example the device driver for 6850 serial interfaces is called “ACIA” and can communicate to any number of serial terminals.

Device driver modules use a standard module header and are given a module type of “device driver” (code $E).The execution offset address in the module header points to a branch table that has a minimum of six (three-byte) entries.Each entry is typically a LBRA to the corresponding subroutine. The File Managers call specific routines in the device driver through this table, passing a pointer to a path decriptor and the hardware control register address in the MPU registers. The branch table looks like:

+0 = Device Initialization Routine
+3 = Read From Device
+6 = Write to Device
+9 = Get Device Status
+$C = Set Device Status
+$F = Device Termination Routine

For a complete description of the parameters passed to these subroutines see the file manager descriptions. Also see the appendicies on writing device drivers.

Device Descriptor Modules

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

Recall that device drivers and file managers both operate on general classes of devices, not specific I/O ports. The device descriptor modules tailor their functions to a specific I/O device. One device descriptor module must exist for each I/O device in the system.

The name of the module is the name the device is known by to the system and user (i.e. it is the device name given in pathlists}. Its format consists of a standard module

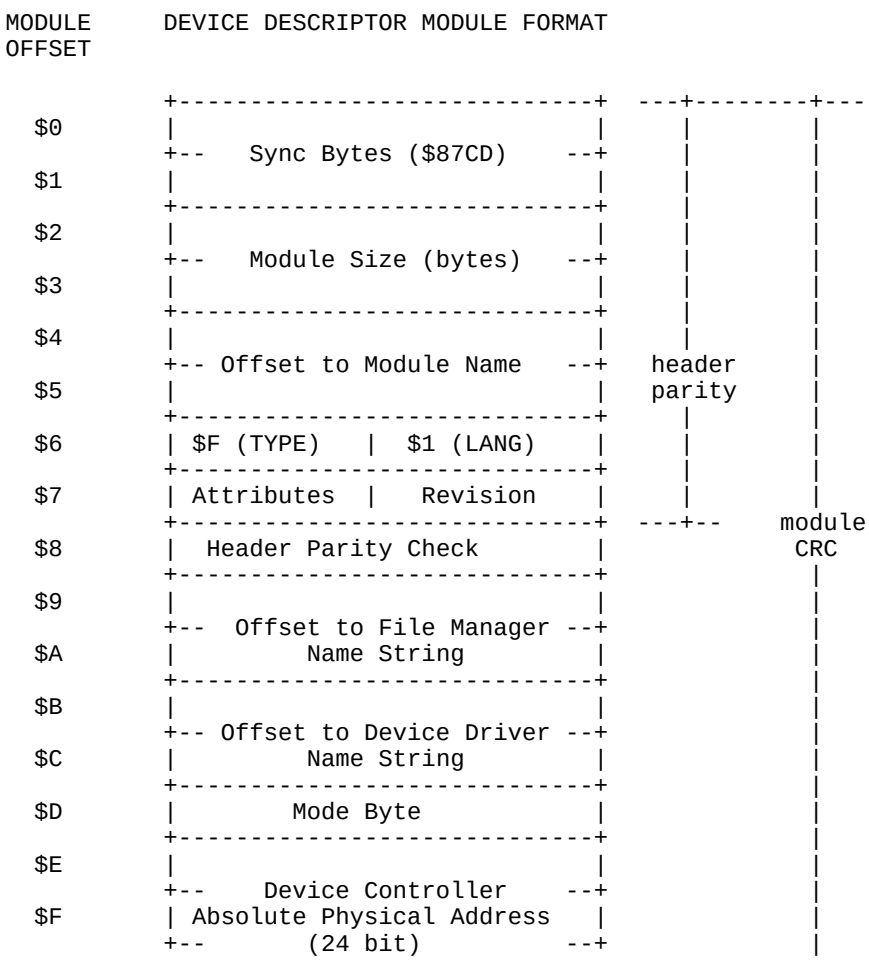
header that has a type “device descriptor” (code $F). The rest of the device descriptor header consists of:

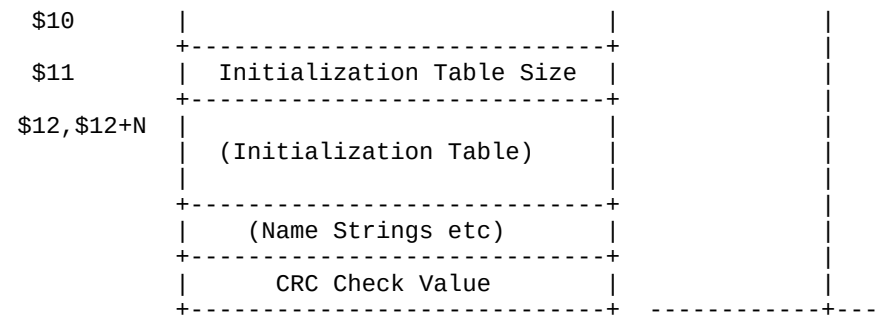
$9,$A =	File manager name string relative address.
$B,$C =	Device driver name string relative address
$D =	Mode/Capabilities. (D S PE PW PR E W R)
$E,$F,$10 =	Device controller absolute physical (24-bit) address
$11 =	Number of bytes (“n” bytes in intialization table)
$12,$12+n =	Initialization table

The initialization table is copied into the “option section” of the path descriptor when a path to the device is opened. The values in this table may be used to define the operating parameters that are changeable by the OS9 I$GetStt and I$SetStt service requests. For example, a terminal’s initialization parameters define which control characters are used for backspace, delete, etc. The maximum size of initialization table which may be used is 32 bytes. If the table is less than 32 bytes long, the remaining values in the path descriptor will be set to zero.

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

The diagram on the next page illustrates the device descriptor module format.





Path Descriptors

Every open path is represented by a data structure called a path descriptor ("PD"). It contains the information required by the file managers and device drivers to perform I/O functions. Path descriptors are exactly 64 bytes long and are dynamically allocated and deallocated by IOMAN as paths are opened and closed.

PDs are INTERNAL data structures that are not normally referenced from user or applications programs. In fact, it is almost impossible to locate a path's PD when OS-9 is in user mode. The description of PDs is mostly of interest to, and presented here for those programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections: the first 10-byte section is defined universally for all file managers and device drivers, as shown below.

Table 5-1. Universal Path Descriptor Definitions

Name	Addr	Size	Description
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1=read 2=write 3=update
PD.CNT	$02	1	Number of paths using this PD
PD.DEV	$03	2	Address of associated device table entry
PD.CPR	$05	1	Requester's process ID
PD.RGS	$06	2	Caller's MPU register stack address
PD.BUF	$08	2	Address of 236-byte data buffer (if used)
PD.FST	$0A	22	Defined by file manager
PD.OPT	$20	32	Reserved for GETSTAT/SETSTAT options

The 22-byte section called "PD.FST" is reserved for and defined by each type of file manager for file pointers, permanent variables, etc.

The 32-byte section called "PD.OPT" is used as an "option" area for dynamically-alterable operating parameters for the file or device. These variables are initialized at the time the path is opened by copying the initialization table contained in the device descriptor module, and can be altered later by user programs by means of the I$GetStt and I$SetStt system calls.

These two sections are defined each file manager's in the assembly language equate file (`SCFDefs` for SCFMAN and `RBFDefs` for RBFMAN).

Chapter 6. Random Block File Manager

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

In the course of normal operation, RBFMAN requests allocation and deallocation of 256-byte data buffers; usually one is required for each open file. When physical I/O functions are necessary, RBFMAN directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks. RBFMAN does not directly deal with physical addresses such as tracks, cylinders, etc. Instead, it passes to device driver modules address parameters using a standard address called a "logical sector number", or "LSN". LSNs are integers in the range of 0 to n-1, where n is the maximum number of sectors on the media. The driver is responsible for translating the logical sector number to actual cylinder/track/sector values.

Because RBFMAN is designed to support a wide range of devices having different performance and storage capacity, it is highly parameter-driven. The physical parameters it uses are stored on the media itself. On disk systems, this information is written on the first few sectors of track number zero. The device drivers also use this information, particularly the physical parameters stored on sector 0. These parameters are written by the "format" program that initializes and tests the media.

Logical and Physical Disk Organization

All mass storage volumes (disk media) used by OS-9 utilize the first few sectors of the volume to store basic identification structure, and storage allocation information.

Logical sector zero (LSN 0) is called the *Identification Sector* which contains description of the physical and logical format of the volume.

Logical sector one (LSN 1) contains an allocation map which indicates which disk sectors are free and available for use in new or expanded files.

The volume's root directory usually starts at logical sector two.

Identification Sector

Logical sector number zero contains a description of the physical and logical characteristics of the volume. These are established by the **format** command program when the media is initialized, the table below gives the OS-9 mnemonic name, byte address, size, and description of each value stored in this sector.

Name	Addr	Size	Description
DD.TOT	$00	3	Total number of sectors on media
DD.TKS	$03	1	Number of sectors per track
DD.MAP	$04	2	Number of bytes in allocation map
DD.BIT	$06	2	Number of sectors per cluster
DD.DIR	$08	3	Starting sector of root directory
DD.OWN	$0B	2	Owner's user number
DD.ATT	$0D	1	Disk attributes
DD.DSK	$05	2	Disk identification (for internal use)
DD.FMT	$10	1	Disk format: density, number of sides
DD.SPT	$11	2	Number of sectors per track

Name	Addr	Size	Description
DD.RES	$13	2	Reserved for future use
DD.BT	$15	3	Starting sector of bootstrap file
DD.BSZ	$18	2	Size of bootstrap file (in bytes)
DD.DAT	$1A	5	Time of creation: Y:M:D:H:M
DD.NAM	$1F	32	Volume name: last char has sign bit set
DD.OPT	$3F	32	Option area

Disk Allocation Map Sector

One sector (usually LSN 1) of the disk is used for the "disk allocation map" that specifies which clusters on the disk are available for allocation of file storage space The address of this sector is always assigned logical sector 1 by the format program DD.MAP specifies the number of bytes in this sector which are actually used in the map.

Each bit in the map corresponds to a cluster of sectors on the disk. The number of sectors per cluster is specified by the "DD.BIT" variable in the identification sector, and is always an integral power of two, i,e., 1, 2, 4, 8, 16, etc. There are a maximum of 4096 bits in the map, so media such as double-density double-sided floppy disks and hard disks will use a cluster size of two or more sectors. Each bit is cleared if the corresponding cluster is available for allocation, or set if the sector is already allocated, non-existent, or physically defective. The bitmap is initially created by the **format** utility program.

File Descriptor Sectors

The first sector of every file is called a "file descriptor", which contains the logical and physical description of the file.. The table below describes the contents of the descriptor.

Name	Addr	Size	Description
FD.ATT	$0	1	File Attributes: D S PE PW PR E W R
FD.OWN	$1	2	Owner's User ID
FD.DAT	$3	5	Date Last Modified; Y M D H M
FD.LNK	$8	1	Link Count
FD.SIZ	$9	4	File Size (number of bytes)
FD.DCR	$D	3	Date Created: Y M D
FD.SEG	$10	240	Segment List: see below

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

The segment list consists of up to 48 five-byte entries that have the size and address of each block of storage that comprise the file in logical order. Each entry has a three-byte logical sector number of the block, and a two-byte block size (in sectors). The entry following the last segment will be zero.

When a file is created, it initially has no data segments allocated to it. Write operations past the current end-of-file (the first write is always past the end-of-file) cause additional sectors to be allocated to the file. If the file has no segments, it is given an initial segment having the number of sectors specified by the minimum allocation entry in the device descriptor, or the number of sectors requested if greater than the

minimum. Subsequent expansions of the file are also generally made in minimum allocation increments. An attempt is made to expand the last segment wherever possible rather than adding a new segment. When the file is closed, unused sectors in the last segment are truncated.

A note about disk allocation: OS-9 attempts to minimize the number of storage segments used in a file. In fact, many files will only have one segment in which case no extra read operations are needed to randomly access any byte on the file. Files can have multiple segments if the free space of the disk becomes very fragmented, or if a file is repeatedly closed, then opened and expanded at some later time. This can be avoided by writing a byte at the highest address to be used on a file before writing any other data.

Directory Files

Disk directories are files that have the "D" attribute set. Directory files contain an integral number of directory entries each of which can bold the name and LSN of a single regular or directory file.

Each directory entry is 32 bytes long, consisting of 29 bytes for the file name followed by a three byte logical sector number of the file's descriptor sector. The file name is left-justified in the field with the sign bit of the last character set. Unused entries have a zero byte in the first file name character position.

Every mass-storage media must have a master directory called the "root directory". The beginning logical sector number of this directory is stored in the identification sector, as previously described.

RBFMAN Definitions of the Path Descriptor.

The table below describes the usage of the file-manager-reserved section of path descriptors used by RBFMAN.

Name	Addr	Size	Description
Universal Section (same for all file managers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Mode (read/write/update)
PD.CNT	$02	1	Number of open images
PD.DEV	$03	2	Address of device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of callers register stack
PD.BUF	$08	2	Buffer address
RBFMAN Path Descriptor Definitions			
PD.SMF	$0A	1	State flags (see next page)
PD.CP	$0B	4	Current logical file position (byte addr)
PD.SIZ	$0F	4	File size
PD.SBL	$13	3	Segment beginning logical sector number
PD.SBP	$16	3	Segment beginning physical sector number
PD.SSZ	$19	2	Segment size
PD.DSK	$15	2	Disk ID (for internal use only)
PD.DTB	$lD	2	Address of drive table

Name	Addr	Size	Description
RBFMAN Option Section Definitions (Copied from device descriptor)			
	$20	1	Device class 0= SCF 1=NSF 2=PIPE 3=SBF
PD.DRV	$21	1	Drive number (0..N)
PD.STP	$22	1	Step rate
PD.TYV	$23	1	Device type
PD.UNS	$24	1	Density capability
PD.CYL	$25	2	Number of cylinders (tracks)
PD.SID	$27	1	Number of sides (surfaces)
PD.VFY	$28	1	0 = verify disk writes
PD.SCT	$29	2	Default number of sectors/track
PD.T0S	$2B	2	Default number of sectors/track (track 0)
PD.ILV	$2D	1	Sector interleave factor
PD.SAS	$2E	1	Segment allocation size
(the following values are *not* copied from the device descriptor)			
PD.ATT	$33	1	File attributes (D S PE PW PR E W R)
PD.FD	$34	3	File descriptor PSN (physical sector #)
PD.DFD	$37	3	Directory file descriptor PSN
PD.DCP	$3A	4	File's directory entry pointer
PD.DVT	$3E	2	Address of device table entry

State Flag (PD.SMF): the bits of this byte are defined as:

bit 0 = set if current buffer has been altered
bit 1 = set if current sector is in buffer
bit 2 = set if descriptor sector in buffer

The first section of the path descriptor is universal for all file managers, the second and third sections are defined by RBFMAN and RBFMAN-type device drivers. The option section of the path descriptor contains many device operating parameters which may be read and/or written by the OS9 I$GetStt and I$SetStt service requests. This section is initialized by IOMAN which copies the initialization table of the device descriptor into the option section of the path descriptor when a path to a device is opened. Any values not determined by this table will default to zero.

RBF Device Descriptor Modules

This section describes the definitions and use of the initialization table contained in device descriptor modules for RBF-type devices.

Module Offset			
0-$11			Standard Device Descriptor Module Header
$12	IT.DTP	RMB 1	DEVICE TYPE (0=SCF 1=RBF 2=PIPE 3=SBF)
$13	IT.DRV	RMB 1	DRIVE NUMBER
$14	IT.STP	RMB 1	STEP RATE
$15	IT.TYP	RMB 1	DEVICE TYPE (See RBFMAN path descriptor)
$16	IT.DNS	RMB 1	MEDIA DENSITY (0 - SINGLE, 1-DOUBLE)

Module Offset			
$17	IT.CYL	RMB 2	NUMBER OF CYLINDERS (TRACKS)
$19	IT.SID	RMB 1	NUMBER OF SURFACES (SIDES)
$1A	IT.VFY	RMB 1	0 = VERIFY DISK WRITES
$1B	IT.SCT	RMB 2	Default Sectors/Track
$1D	IT.T0S	RMB 2	Default Sectors/Track (Track 0)
$1F	IT.ILV	RMB 1	SECTOR INTERLEAVE FACTOR
$20	IT.SAS	RMB 1	SEGMENT ALLOCATION SIZE

IT.DRV - This location is used to associate a one byte integer with each drive that a controller will handle. The drives for each controller should be numbered 0 to n-1, where n is the maximum number of drives the controller can handle.

IT.STP - (Floppy disks) This location sets the head stepping rate that will be used with a drive. The step rate should be set to the fastest value that the drive is capable of to reduce access time. The actual values stored depended on the specific disk controller and disk driver module used. Below are the values which are used by the popular Western Digital floppy disk controller IC:

Step Code	FD1771		FD179X Family	
	5"	8"	5"	8"
0	40ms	20ms	30ms	15ms
1	20ms	10ms	20ms	10ms
2	12ms	6ms	12ms	6ms
3	12ms	6ms	6ms	3ms

IT.TYP - Device type (All types)
bit 0 -- 0 = 5" floppy disk
1 = 8" floppy disk
bit 6 -- 0 = Standard OS-9 format
1 = Non-standard format
bit 7 -- 0 = Floppy disk
1 = Hard disk

IT.DNS - Density capabilities (Floppy disk only)
bit 0 -- 0 = Single bit density (FM)
1 = Double bit density (MFM)

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

IT.SAS - This value specifies the minimum number of sectors to be allocated at any one time.

RBF-type Device Drivers

An RBF type device driver module contains a package of subroutines that perform sector oriented I/O to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. IOMAN will allocate a static storage area for each device (which may control several drives). The size of the storage area is given in the device driver module header. Some of this storage area will be used by

IOMAN and RBFMAN, the device driver is free to use the remainder in any manner. This static storage is used as follows:

Table 6-1. Static Storage Definitions

Offset		ORG 0	
0	V.PAGE	RMB 1	PORT EXTENDED ADDRESS (A20 - A16)
1	V.PORT	RMB 2	DEVICE BASE ADDRESS
3	V.LPRC	RMB 1	LAST ACTIVE PROCESS ID
4	V.BUSY	RMB 1	ACTIVE PROCESS ID (0 = NOT BUSY)
5	V.WAKE	RMB 1	PROCESS ID TO REAWAKEN
	V.USER	EQU .	END OF OS9 DEFINITIONS
6	V.NDRV	RMB 1	NUMBER OF DRIVES
	DRVBEG	EQU .	BEGINNING OF DRIVE TABLES
7	TABLES	RMB DRVMEM*N	RESERVE N DRIVE TABLES
		RMB 1	
	FREE	EQU	FREE FOR DRIVER TO USE

NOTE: V.PAGE through V.USER are predefined in the `OS9Defs` file. V.NDRV, DRVBEG, DRVMEM are predefined in the `RBFDefs` file.

V.PAGE, V.PORT These three bytes are defined by IOMAN as the 24-bit device address.

V.LPRC This location contains the process ID of the last process to use the device. Not used by RBF-type device drivers.

V.BUSY This location contains the process ID of the process currently using the device. Defined by RBFMAN.

V.WAKE This location contains the process-ID of any process that is waiting for the device to complete I/O (0 = NO PROCESS WAITING). Defined by device driver.

V.NDRV This location contains the number of drives that the controller can use. Defined by the device driver as the maximum number of drives that the controller can work with. RBFMAN will assume that there is a drive table for each drive. Also see the driver INIT routine in this section.

TABLES This area contains one table for each drive that the controller will handle (RBFMAN will assume that there are as many tables as indicated by V.NDRV). Some time after the driver INIT routine has been called, RBFMAN will issue a request for the driver to read the identification sector (logical sector zero) from a drive. At this time the driver will initialize the corresponding drive table by copying the first part of the identification sector (up to DD.SIZ) into it, Also see the “Identification Sector” section of this manual. The format of each drive table is as given below:

Offset		ORG 0	
$00	DD.TOT	RMB 3	Total number of sectors on media
$03	DD.TKS	RMB 1	Number of sectors per track
$04	DD.MAP	RMB 2	Number of bytes in allocation map
$06	DD.BIT	RMB 2	Number of sectors per cluster
$08	DD.DIR	RMB 3	Starting sector of root directory
$0B	DD.OWN	RMB 2	Owner’s user number

Offset		**ORG 0**	
$0D	DD.ATT	RMB 1	Disk attributes
$05	DD.DSK	RMB 2	Disk identification
$10	DD.FMT	RMB 1	Disk format: density, number of sides
$11	DD.SPT	RMB 2	Number of sectors per track
$13	DD.RES	RMB 2	Reserved for future use
	DD.SIZ	EQU .	
$15	V.TRAK	RMB 2	Current Track Number
$17	V.BMB	RMB 1	Bit-map Use Flag
$18	DRVMEM	EQU .	Size of Each Drive Table

DD.TOT This location contains the total number of sectors contained on the disk.

DD.TKS This location contains the track size (in sectors).

DD.MAP This location contains the number of bytes in the disk allocation bit map.

DD.BIT This location contains the number of sectors that each bit represents in the disk allocation bit map.

DD.DIR This location contains the logical sector number of the disk root directory.

DD.OWN This location contains the disk owner's user number.

DD.APT This location contains the disk access permission attributes as defined below:

BIT 7 - D (DIRECTORY IF SET)
BIT 6 - S (SHARABLE IF SET)
BIT 5 - PX (PUBLIC EXECUTE IF SET)
BIT 4 - PW (PUBLIC WRITE IF SET)
BIT 3 - PR (PUBLIC READ IF SET)
BIT 2 - X (EXECUTE IF SET)
BIT 1 - W (WRITE IF SET).
BIT 0 - R (READ IF SET)

DD.DSK This location contains a pseudo random number which is used to identify a disk so that OS-9 may detect when a disk is removed from the drive and another inserted in its place.

DD.FMT DISK FORMAT:

BIT B0 - SIDE
  0 = SINGLE SIDED
  1 = DOUBLE SIDED

BIT B1 - DENSITY
  0 = SINGLE DENSITY
  1 = DOUBLE DENSITY

BIT B2 - TRACK DENSITY
  0 = SINGLE (48 TPI)
  1= DOUBLE (96 TPI)

DD.SPT Number of sectors per track (track zero may use a different value, specified by IT.T0S in the device descriptor).

DD.RES RESERVED FOR FUTURE USE

V.TRAK This location contains the current track which the head is on and is updated by the driver.

V.BMB This location is used by RBFMAN to indicate whether or not the disk allocation bit map is currently in use (0 = not in use). The disk driver routines must not alter this location.

RBFMAN Device Drivers

As with all device drivers, RBFMAN-type device drivers use a standard executable memory module format with a module type of "device driver" (CODE $E). The execution offset address in the module header points to a branch table that has six three byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY   LBRA    INIT    INITIALIZE DRIVE

        LBRA    READ    READ SECTOR
        LBRA    WRITE   WRITE SECTOR
        LBRA    GETSTA  GET STATUS
        LBRA    SETSTA  SET STATUS
        LBRA    TERM    TERMINATE DEVICE
```

Each subroutine should exit with the condition code register C bit cleared if no error occurred. Otherwise the C bit should be set and an appropriate error code returned in the B register. Below is a description of each subroutine, its input parameters, and its output parameters.

NAME: INIT

NAME:	INIT
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE (Y) = ADDRESS OF THE DEVICE DESCRIPTOR MODULE
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	INITIALIZE DEVICE AND ITS STATIC STORAGE AREA

1. If disk writes are verified, use the F$SRqMem service request to allocate a 256 byte buffer area where a sector may be read back and verified after a write.
2. Initialize the device permanent storage. For floppy disk controller typically this consists of initializing V.NDRV to the number of drives that the controller will work with, initializing DD.TOT in the drive table to a non-zero value so that sector zero may be read or written to, and initializing V.TRAK to $FF so that the first seek will find track zero.
3. Place the IRQ service routine on the IRQ polling list by using the OS9 F$IRQ service request.
4. Initialize the device control registers (enable interrupts if necessary).

NOTE: Prior to being called, the device permanent storage will be cleared (set to zero) except for V.PAGE and V.PORT which will contain the 24 bit device address. The driver should initialize each drive table appropriately for the type of disk the driver expects to be used on the corresponding drive.

NAME: READ

NAME:	READ
INPUT:	(U) = ADDRESS OF THE DEVICE STATIC STORAGE (Y) = ADDRESS OF THE PATH DESCRIPTOR (B) = NSB OF DISK LOGICAL SECTOR NUMBER (X) = LSB's OF DISK LOGICAL SECTOR NUMBER
OUTPUT:	SECTOR IS RETURNED IN THE SECTOR BUFFER
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	READ A 256 BYTE SECTOR

Read a sector from the disk and place it in the sector buffer (256 byte). Below are the things that the disk driver must do:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.,DRV in the path descriptor.

3. Compute the physical disk address from the logical sector.

number.

4. Initiate the read operation.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete (the IRQ service routine is responsible for sending a wake up signal). After awakening, test V.WAKE to see if it is clear, if not, go back to sleep.

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

NOTE 1: Whenever logical sector zero is read, the first part of this sector must be copied into the proper drive table (get the drive number from PD.DRV in the path descriptor). The number of bytes to copy is DD.SIZ.

NOTE 2: The drive number (PD.DRv) should be used to compute the offset to the corresponding drive table as follows:

```
LDA PD.DRV.Y Get drive number
LDB #DRVMEM Get size of a drive table
MUL
LEAX DRVBEG,U Get address of first table
LEAX D,X          Compute address of table N
```

NAME: WRITE

NAME:	WRITE

INPUT:	(U) = ADDRESS OF THE DEVICE STATIC STORAGE (Y) = ADDRESS OF THE PATH DESCRIPTOR (B) = MSB OF DISK LOGICAL SECTOR NUMBER (X) = LSB's OF DISK LOGICAL SECTOR NUMBER
OUTPUT:	THE SECTOR BUFFER IS WRITTEN OUT TO DISK
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	WRITE A SECTOR

Write the sector buffer (256 bytes) to the disk. Below are the things that a disk driver must do:

1. Get the sector buffer address from PD.BUF in the path descriptor.

2. Get the drive number from PD.DRV in the path descriptor.

3. Compute the physical disk address from the logical sector number.

4. Initiate the write operation.

5. Copy V.BUSY to V.WAKE, then go to sleep and wait for the I/O to complete (the IRQ service routine is responsible for sending the wakeup signal). After awakening, test V.WAKE to see if it is clear, if it is not, then go back to sleep. If the disk controller can not be interrupt-driven, it will be necessary to perform a programmed I/O transfer.

6. If PD.VFY in the path descriptor is equal to zero, read the sector back in and verify that it was written correctly. This usually does not involve a compare of the data.

NOTE 1: If disk writes are to be verified, the INIT routine must request the buffer where the sector may be placed when it is read back in. Do not copy sector zero into the drive table when it is read back to be verified.

NOTE 2: Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the READ routine.

NAME: GETSTA PUTSTA

NAME:	GETSTA/PUTSTA
INPUT:	(U) = ADDRESS OF THE DEVICE STATIC STORAGE AREA (Y) = ADDRESS OF THE PATH DESCRIPTOR (A) = STATUS CODE
OUTPUT:	(DEPENDS UPON THE FUNCTION CODE)
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	GET/SET DEVICE STATUS

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

It may be necessary to examine or change the register stack which contains the values of MPU registers at the time of the I$GetStt or I$SetStt service request. The address of the register stack may be found in PD.RGS, which is located in the path descriptor, . The following offsets may be used to access any particular value in the register stack:

OFFSET	MNEMONIC			MPU REGISTER
$0	R$CC	RMB	1	CONDITIONS CODE REGISTER
$1	R$D	EQU	.	D REGISTER
$1	R$A	RMB	1	A REGISTER
$2	R$B	RMB	1	B REGISTER
$3	R$DP	RMB	1	DP REGISTER
$4	R$X	RMB	2	X REGISTER
$6	R$Y	RMB	2	Y REGISTER
$8	R$U	RMB	2	U REGISTER
$A	R$PC	RMB	2	PROGRAM COUNTER

NAME: TERM

NAME:	TERM
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE AREA
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	TERMINATE DEVICE

This routine is called when a device is no longer in use in the system, which is defined to be when the link count of its device descriptor module becomes zero). The TERM routine must:

1. Wait until any pending I/O has completed.

2. Disable the device interrupts.

3. Remove the device from the IRQ polling list.

4. If the INIT routine reserved a 256 byte buffer for verifying disk writes, return the memory with the F$Mem service request.

NAME: IRQ SERVICE ROUTINE

NAME:	IRQ SERVICE ROUTINE
FUNCTION:	SERVICE DEVICE INTERRUPTS

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

1. Service device interrupts.

2. When the I/O is complete, the IRQ service routine should send a wake up signal to the process whose process ID is in V.WAKE

Also clear V.WAKE as a flag to the mainline program that the IRQ has indeed occurred.

NOTE: When the IRQ service routine finishes servicing an interrupt it must clear the array and exit with an RTS instruction.

NAME: BOOT (Bootstrap Module)

NAME:	TERM
INPUT:	None.
OUTPUT:	(U) = SIZE OF THE BOOT FILE (in bytes) (X) = AD- DRESS OF WHERE THE BOOT FILE WAS LOADED IN MEM- ORY
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	LOAD THE BOOT FILE INTO MEMORY FROM MASS-STORAGE

NOTE: The BOOT module is *not* part of the disk driver. It is a separate module which is normally co-resident with the "OS9P2" module in the system firmware.

The bootstrap module contains one subroutine that loads the bootstrap file and some related information into memory, it uses the standard executable module format with a module type of "system" (code $C). The execution offset in the module header contains the offset to the entry point of this subroutine.

It obtains the starting sector number and size of the `OS9Boot` file from the identification sector (LSN 0). OS-9 is called to allocate a memory area large enough for the boot file, and then it loads the boot file into this memory area.

1. Read the identification sector (sector zero) from the disk. BOOT must pick its own buffer area. The identification sector contains the values for DD.BT (the 24 bit logical sector number of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes). For a full description of the identification sector. See the Section called *Disk Allocation Map Sector*.

2. After reading the identification sector into the buffer, get the 24 bit logical sector number of the bootstrap file from DD.BT.

3. Get the size (in bytes) of the bootstrap file from DD.BSZ. The boot is contained in one logically contiguous block beginning at the logical sector specified in DD.BT and extending for (DD.BSZ/256+1) sectors.

4. Use the OS9 F$SRqMem service request to request the memory area where the boot file will be loaded into.

5. Read the boot file into this memory area.

6. Return the size of the boot file and its location.

Chapter 7. Sequential Character File Manager

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

Standard OS-9 systems are supplied with SCFMAN and two SCF-type device driver modules: ACIA, which run 6850 serial interfaces, and PIA, which drives a 6821-type parallel interface for printers.

SCFMAN Line Editing Functions

I$Read and I$Write service requests (which correspond to Basic09 GET and PUT statements) to SCFMAN-type devices pass data to/from the device without any modification, except that keyboard interrupt, keyboard abort, and pause character are filtered out of the input (editing is disabled if the corresponding character in the path descriptor contains a zero). In particular, carriage returns are not automatically followed by line feeds or nulls, and the high order bits are passed as sent/received.

I$ReadLn and I$WritLn service requests (which correspond to Basic09 INPUT, PRINT, READ and WRITE statements) to SCFMAN-type devices perform full line editing of all functions enabled for the particular device. These functions are initialized when the device is first used by copying the option table from the device descriptor table associated with the specific device. They may be altered anytime afterwards from assembly language programs using the I$SetStt and I$GetStt service requests, or from the keyboard using the **tmode** command. Also, all bytes transfered in this mode will have the high order bit cleared.

The following path descriptor values control the line editing functions:

If PD.UPC <> 0 bytes input or output in the range “a..z” are made “A..Z”

If PD.EKO <> 0, input bytes are echoed, except that undefined control characters in the range $0..$1F print as “.”

If PD.ALF <> 0, carriage returns are automatically followed by line feeds.

If PD.NUL <> 0, After each CR/LF a PD.NUL “nulls” (always $00) are sent.

If PD.PAU <> 0, Auto page pause will occur after every PD.PAU lines since the last input.

If PD.BSP <> 0, SCF will recognize PD.BSP as the “input” backspace character, and will echo PD.BSE (the backspace echo character) if PD.BSO = 0, or PD.BSE, space, PD.BSE if PD.BSO <> 0.

If PD.DEL <> 0, SCF will recognize PD.DEL the delete line character (on input), and echo the backspace sequence over the entire line if PD.DLO = 0, or echo CR/LF if PD.DLO <> 0.

PD.EOR defines the end of record character. This is the last character an each line entered (I$ReadLn), and terminates the output (I$WritLn) when this character is sent. Normally PD.EOR will be set to $0D. If it is set to zero, SCF’s I$ReadLn will NEVER terminate, unless an EOF occurs.

If PD.EOF <> 0, it defines the end of file character. SCFMAN will return an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input. It can be disabled by setting its value to zero.

If PD.RPR <> 0, SCF (I$ReadLn) will, upon receipt of this character, echo a carriage return [optional line feed], and then reprint the current line.

If PD.DUP <> 0, SCF (I$ReadLn) will duplicate whatever is in the input buffer through the first "PD.EOR" character.

If PD.PSC <> 0, output is suspended before the next "PD.EOR" character when this character is input. This will also delete any "type ahead" input for I$ReadLn.

If PD.INT <> 0, and is received on input, a keyboard interrupt signal is sent to the last user of this path. Also it will terminate the current I/O request (it any) with an error identical to the keyboard interrupt signal code. This location normally is set to a **control**+**C** character.

If PD.QUT <> 0, and is received on input, a keyboard abort signal is sent to the last user of this path. Also it will terminate the current I/O request (if any) with an error code identical to the keyboard interrrupt signal code. This location is normally set to a **control**+**Q** character.

If PD.OVF <> 0, It is echoed when I$ReadLn has satisfied its input byte count without finding a "PD.EOR" character.

NOTE: It is possible to disable most of these special editing functions by setting the corresponding control character in the path descriptor to zero by using the I$SetStt service request, or by running the **tmode** utility. A more permanent solution may be had by setting the corresponding control character value in the device descriptor module to zero.

Device descriptors may be inspected to determine the default settings for these values for specific devices.

SCFMAN Definitions of The Path Descriptor

The table below describes the path descriptors used by SCFMAN and SCFMAN-type device drivers.

Name	Offset	Size	Description
Universal Section (same for all file managers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Mode (read/write/update)
PD.CNT	$02	1	Number of open images
PD.DEV	$03	2	Address of device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of callers register stack
PD.BUF	$08	2	Buffer address
SCFMAN Path Descriptor Definitions			
PD.DV2	$0A	2	Device table addr of 2nd (echo) device
PD.RAW	$0C	1	Edit flag: 0=raw mode, 1=edit mode
PD.MAX	$0D	2	Headline maximum character count
PD.MIN	$0F	1	Devices are "mine" if cleared
PD.STS	$10	2	Status routine module address
PD.STM	$12	2	Reserved for status routine
SCFMAN Option Section Definition			
	$20	1	Device class 0=SCF 1=RBF 2=PIPE 3=SBF
PD.UPC	$21	1	Case (0=BOTH, 1=UPPER ONLY)
PD.BSO	$22	1	Backsp (0=BSE, 1=BSE SP BSE)

Name	Offset	Size	Description
PD.DLO	$23	1	Delete (0 = BSE over line, 1=CR LF)
PD.EKO	$24	1	Echo (0=no echo)
PD.ALF	$25	1	Auto LF (0=no auto LF)
PD.NUL	$26	1	End of line null count
PD.PAU	$27	1	Pause (0= no end of page pause)
PD.PAG	$28	1	Lines per page
PD.BSP	$29	1	Backspace character
PD.DEL	$2A	1	Delete line character
PD.EOR	$25	1	End of record character (read only)
PD.EOF	$2C	1	End of file character (read only)
PD.RPR	$2D	1	Reprint line character
PD.DUP	$25	1	Duplicate last line character
PD.PSC	$2F	1	Pause character
PD.INT	$30	1	Keyboard interrupt character (CTL C)
PD.QUT	$31	1	Keyboard abort character (CTL Q)
PD.BSE	$32	1	Backspace echo character (BSE)
PD.OVF	$33	1	Line overflow character (bell)
PD.PAR	$34	1	Device initialization value (parity)
PD.BAU	$35	1	Software settable baud rate
PD.D2P	$36	2	Offset to 2nd device name string
PD.STN	$38	2	Offset of status routine name
PD.ERR	$3A	1	Most recent I/O error status

The first section is universal for all file managers, the second and third section are specific for SCFMAN and SCFMAN-type device drivers. The option section of the path descriptor contains many device operating parameters whicb may be read or written by the OS9 I$GetStt or I$SetStt service requests. IOMAN initializes this section when a path is opened to a device by copying the corresponding device descriptor initialization table. Any values not determined by this table will default to zero.

Special editing functions may be disabled by setting the corresponding control character value to zero.

SCF Device Descriptor Modules

Device descriptor modules for SCF-type devices contain the device address and an initialization table which defines inital values for the I/O editing features, as listed below.

MODULE OFFSET		ORG $12	
	TABLE	EQU .	BEGINING OF OPTION TABLE
$12	IT.DVC	RMB 1	DEVICE CLASS (0=SCF 1=RBF 2=PIPE 3=SBF)
$13	IT.UPC	RMB 1	CASE (0=BOTH, 1=UPPER ONLY)
$14	IT.BSO	RMB 1	BACK SPACE (0=BSE, 1=BSE,SP,BSE)
$15	IT.DLO	RMB 1	DELETE (0=BSE OVER LINE, 1=CR)

MODULE OFFSET		ORG $12	
$16	IT.EKO	RMB 1	ECHO (0=NO ECHO)
$17	IT.ALF	RMB 1	AUTO LINE FEED (0= NO AUTO LF)
$18	IT.NUL	RMB 1	END OF LINE NULL COUNT
$19	IT.PAU	RMB 1	PAUSE (0= NO END OF PAGE PAUSE)
$1A	IT.PAG	RMB 1	LINES PER PAGE
$1B	IT.BSP	RMB 1	BACKSPACE CHARACTER
$1C	IT.DEL	RMB 1	DELETE LINE CHARACTER
$1D	IT.EOR	RMB 1	END OF RECORD CHARACTER
$1E	IT.EOF	RMB 1	END OF FILE CHARACTER
$1F	IT.RPR	RMB 1	REPRINT LINE CHARACTER
$20	IT.DUP	RMB 1	DUP LAST LINE CHARACTER
$21	IT.PSC	RMB 1	PAUSE CHARACTER
$22	IT.INT	RMB 1	INTERRUPT CHARACTER
$23	IT.QUT	RMB 1	QUIT CHARACTER
$24	IT.BSE	RMB 1	BACKSPACE ECHO CHARACTER
$25	IT.OVF	RMB 1	LINE OVERFLOW CHARACTER (BELL)
$26	IT.PAR	RMB 1	INITIALIZATION VALUE (PARITY)
$27	IT.BAU	RMB 1	BAUD RATE
$28	IT.D2P	RMB 2	ATTACHED DEVICE NAME STRING OFFSET
$2A	IT.STN	RMB 2	OFFSET TO STATUS ROUTINE
$2C	IT.ERR	RMB 1	INITIAL ERROR STATUS

NOTES:

SCF editing functions will be “turned off” if the corresponding special character is a zero. For example, it IT.EOF was a zero, there would be no end of file character.

IT.PAR is typically used to intitialize the device’s control register when a path is opened to it.

SCF Device Driver Storage Definitions

An SCFMAN-type device driver module contains a package of subroutines that perform raw I/O transfers to or from a specific hardware controller. These modules are usually reentrant so that one copy of the module can simultaneously run several different devices that use identical I/O controllers. For each “incarnation” of the driver, IOMAN will allocate a static storage area for that device. The size of the storage area is given in the device driver module header. Some of this storage area will be used by IOMAN and SCFMAN, the device driver is free to use the remainder in any way (typically as variables and butters). This static storage is defined as:

OFFSET		ORG 0	
$0	V.PAGE	RMB 1	PORT EXTENDED ADDRESS
$1	V.PORT	RMB 2	DEVICE BASE ADDRESS
$3	V.LPRC	RMB 1	LAST ACTIVE PROCESS ID
$4	V.BUSY	RMB 1	ACTIVE PROCESS ID (0 NOT BUSY)

OFFSET		**ORG 0**	
$5	V. WAKE	RMB 1	PROCESS ID TO REAWAKEN
	V. USER	EQU .	END OF OS9 DEFINITIONS
$6	V.TYPE	RMB 1	DEVICE TYPE OR PARITY
$7	V.LINE	RMB 1	LINES LEFT TILL END OF PAGE
$8	V.PAUS	RMB 1	PAUSE REQUEST (0 = NO PAUSE)
$9	V.DEV2	RMB 2	ATTACHED DEVICE STATIC STORAGE
$B	V. INTR	RMB 1	INTERRUPT CHARACTER
$C	V.QUIT	RMB 1	QUIT CHARACTER
$D	V.PCHR	RMB 1	PAUSE CHARACTER
$E	V. ERR	RMB 1	ERROR ACCUMULATOR
$F	V.SCF	EQU .	END OF SCFMAN DEFINITIONS
	FREE	EQU .	FREE FOR DEVICE DRIVER TO USE

V.PAGE, V.PORT These three bytes are defined by IOMAN to be the 24 bit device address.

V.LPRC This location contains the process ID of the last process to use the device. The IRQ service routine is responsible for sending this process the proper signal in case a "QUIT" character or an "INTERRUPT" character is recieved. Defined by SCFMAN.

V. BUSY This location contains the process ID of the process currently using the device (zero if it is not being used). This is used by SCFMAN to prevent more than one process from using the device at the same moment. Defined by SCFMAN.

V.WAKE This location contains the process ID of any process that is waiting for the device to complete I/O (or zero if there is none waiting). The interrupt service routine should check this location to see if a process is waiting and if so, send it a wake up signal. Defined by the device driver.

V.TYPE This location contains any special characteristics of a device. It is typically used as a value to initialize the device control register, for parity etc. It is defined by SCFMAN which copies its value from PD.PAR in the path descriptor.

V.LINE This location contains the number of lines left till end of page. Paging is handled by SCFMAN and not by the device driver.

V.PAUS This location is a flag used by SCFMAN to indicate that a pause character has been recieved. Setting its value to anything other than zero will cause SCFMAN to stop transmitting characters at the end of the next line. Device driver input routines must set V.PAUS in the ECHO device's static storage area. SCFMAN will check this value in the ECHO device's static storage when output is sent.

V.DEV2 This location contains the address of the ECHO (attached) device's static storage area. Typically the device and the attached device are one and the same. However they may be different as in the case of a keyboard and a memory mapped video display. Defined by SCFMAN.

V.INTR Keyboard interrupt character. It is defined by SCFMAN, which copies its value from PD.INT in the path descriptor.

V.QUIT Keyboard abort character. It is defined by SCFMAN which copies its value from PD.QUT in the path descriptor.

V.PCHR Pause character. It is defined by SCFMAN which copies its value from PD.PSC in the path descriptor.

V.ERR This location is used to accumulate I/O errors. Typically it is used by the IRQ service routine to record errors so that they may be reported later when SCFMAN calls one of the device driver routines.

SCFMAN Device Driver Subroutines

As with all device drivers. SCFMAN device drivers use a standard executable memory module format with a module type of "device driver" (CODE $5). The execution offset address in the module header points to a branch table that has six three byte entries. Each entry is typically a LBRA to the corresponding subroutine. The branch table is as follows:

```
ENTRY    LBRA    INIT      INITIALIZE DEVICE

         LBRA    READ      READ CHARACTER
         LBRA    WRITE     WRITE CHARACTER
         LBRA    GETSTA    GET DEVICE STATUS
         LBRA    SETSTA    SET DEVICE STATUS
         LBRA    TERM      TERMINATE DEVICE
```

Each subroutine should exit with the condition code register C bit cleared it no error occured. Otherwise the C bit should be set and an appropriate error code returned in the B register. Below is a description of each subroutine, its input parameters and its output parameters.

NAME: INIT

NAME:	INIT
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE (Y) = ADDRESS OF DEVICE DESCRIP- TOR MODULE
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	INITIALIZE DEVICE AND ITS STATIC STORAGE

1. Initialize the device static storage.

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

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

NOTE: Prior to being called, the device static storage will be cleared (set to zero) except for V.PAGE and V.PORT which will contain the 24 bit device address. There is no need to initialize the portion of static storage used by IOMAN and SCFMAN.

NAME: READ

NAME:	READ
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE (Y) = ADDRESS OF PATH DESCRIPTOR
OUTPUT:	(A) = CHARACTER READ
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	GET NEXT CHARACTER

This routine should get the next character from the input buffer. If there is no data ready, this routine should copy its process ID from V.BUSY into V.WAKE and then use the F$Sleep service request to put itself to sleep.

Later when data is recieved, the IRQ service routine will leave the data in a buffer, then check V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine should send a wakeup signal to it.

NOTE: Data buffers are *not* automatically allocated. It any are used, they should be defined in the device's static storage area.

NAME: WRITE

NAME:	WRITE
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE (Y) = ADDRESS OF THE PATH DESCRIPTOR (A) = CHAR TO WRITE
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	OUTPUT A CHARACTER

This routine places a data byte into an output buffer and enables the device output interrupts. It the data buffer is already full, this routine should copy its process ID from V.BUSY into V.WAKE and then put itself to sleep.

Later when the IRQ service routine transmits a character and makes room for more data in the buffer, it will check V.WAKE to see if there is a process waiting for the device to complete I/O. It there is, it will send a wake up signal to that process.

NOTE: This routine must ensure that the IRQ service routine will start up when data is placed into the buffer. After an interrupt is generated the IRQ service routine will continue to transmit data until the data butter is empty, and then it will disable the device's "ready to transmit" interrupts.

NOTE: Data buffers are *not* automatically allocated. If any are used, they should be defined in the device's static storage.

NAME: GETSTA/SETSTA

NAME:	GETSTA/SETSTA
INPUT:	(U) = ADDRESS OF DEVICE STATIC STORAGE (Y) = ADDRESS OF PATH DESCRIPTOR (A) = STATUS CODE
OUTPUT:	DEPENDS UPON FUNCTION CODE
FUNCTION:	GET/SET DEVICE STATUS

This routine is a wild card call used to get (set) the device parameters specified in the I$GetStt and I$SetStt service requests. Currently all of the function codes defined by Microware for SCF-type devices are handled by IOMAN or SCFMAN. Any codes not defined by Microware will be passed to the device driver.

It may be necessary to examine or change the register packet which contains the values of the 6809 registers at the time the OS9 service request was issued. The address of the register packet may be found in PD.RGS, which is located in the path descriptor. The following offsets may be used to access any particular value in the register packet:

OFFSET	MNEMONIC			MPU REGISTER
$0	R$CC	RMB	1	CONDITIONS CODE REGISTER
$1	R$D	EQU	.	D REGISTER
$1	R$A	RMB	1	A REGISTER
$2	R$B	RMB	1	B REGISTER
$3	R$DP	RMB	1	DP REGISTER
$4	R$X	RMB	2	X REGISTER
$6	R$Y	RMB	2	Y REGISTER
$8	R$U	RMB	2	U REGISTER
$A	R$PC	RMB	2	PROGRAM COUNTER

NAME. TERM

NAME:	TERM
INPUT:	(U) = PTR TO DEVICE STATIC STORAGE
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	TERMINATE DEVICE

This routine is called when a device is no longer in use, defined as when its device descriptor module's link count becomes zero). It must perform the following:

1. Wait until the output buffer has been emptied (by the IRQ service routine)

2. Disable device interrupts.

3. Remove device from the IRQ polling list.

NOTE: Static storage used by device drivers is never returned to the free memory pool. Therefore, it is desirable to NEVER terminate any device that might be used again. Modules contained in the `Boot` file will NEVER be terminated.

NAME: IRQ SERVICE ROUTINE

NAME:	IRQ SERVICE ROUTINE
FUNCTION:	SERVICE DEVICE INTERRUPTS

Although this routine is not included in the device drivers branch table and not called directly from SCFMAN, it is an important routine in device drivers. The main things that it does are:

1. Service the device interrupts (recieve data from device or send data to it). This routine should put its data into and get its data from buffers which are defined in the device static storage.
2. Wake up any process waiting for I/O to complete by checking to see if there is a process ID in V.WAKE (non-zero) and it so send a wakeup signal to that process.
3. If the device is ready to send more data and the output buffer is emoty, disable the device's "ready to transmit" interrupts.

4. If a pause character is recieved, set V.PAUS in the attached device static storage to a non-zero value. The address of the attached device static storage is in V.DEV2.

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

Chapter 8. Assembly Language Programming Techniques

There are four key rules for programmers writing OS-9 assembly language programs:

1. All programs *must* use position-independent-code (PIC). OS9 selects load addresses based on available memory at run-time. There is no way to force a program to be loaded at a specific address.
2. All programs must use the standard OS-9 memory module formats or they cannot be loaded and run. Programs must not use self-modifying code. Programs must not change anything in a memory module or use any part of it for variables.
3. Storage for all variables and data structures must be within a data area which is assigned by OS-9 at run-time, and is separate from the program memory module.
4. All input and output operations should be made using OS-9 service request calls.

Fortunately, the 6809's versatile addressing modes make the rules above easy to follow,. The OS-9 Assembler also helps because it has special capabilities to assist the programmer in creating programs and memory modules for the OS-9 execution environment.

How to Write Position-Independent Code

The 6809 instruction set was optimized to allow efficient use of Position Independent Code (PIC). The basic technique is to always use PC-relative addressing; for example BRA, LBRA, BSR and LBSR. Get addresses of constants and tables using LEA instructions instead of load immediate instructions. If you use dispatch tables, use tables of RELATIVE, not absolute, addresses.

INCORRECT	CORRECT
LDX #CONSTANT	LEAX CONSTANT,PCR
JSR SUBR	BSR SUBR or LBSR SUBR
JMP LABEL	BRA LABEL or LBRA LABEL

Addressing Variables and Data Structures

Programs executed as processes (by F$Fork and F$Chain system calls or by the `Shell`) are assigned a RAM memory area for variables, stacks, and data structures at execution-time. The addresses cannot be determined or specified ahead of time. However, a minimum size for this area is specified in the program's module header. Again, thanks to the 6809's full compliment of addressing modes this presents no problem to the OS-9 programmer.

When the program is first entered, the Y register will have the address of the top of the process' data memory area. If the creating process passed a parameter area, it will be located from the value of the SP to the top of memory (Y), and the D register will contain the parameter area size in bytes. If the new process was called by the `shell`, the parameter area will contain the part of the `shell` command line that includes the argument (parameter) text. The U register will have the lower bound of the data memory area, and the DP register will contain its page number.

The most important rule is to *not use extended addressing!* Indexed and direct page addressing should be used exclusively to access data area values and structures. Do not use program-counter relative addressing to find addresses in the data area, but

do use it to refer to addresses within the program area.

The most efficient way to handle tables, buffers, stacks, etc., is to have the program's initialization routine compute their absolute addresses using the data area bounds passed by OS-9 in the registers. These addresses can then be saved in the direct page where they can be loaded into registers quickly, using short instructions. This technique has advantages: it is faster than extended addressing, and the program is inherently reentrant.

Stack Requirements

Because OS-9 uses interrupts extensively, and also because many reentrant 6809 programs use the MPU stack for local variable storage, a generous stack should be maintained at all times. The recommended minimum is approximately 200 bytes.

Interrupt Masks

User programs should keep the condition codes register F (FIRQ mask) and I (IRQ mask) bits off. They can be set during critical program sequences to avoid task-switching or interrupts, but this time should be kept to a minimum. If they are set for longer than a tick period, system timekeeping accuracy may be affected. Also, some Level Two systems will abort programs having a set IRQ mask.

Writing Interrupt-driven Device Drivers

OS-9 programs do not use interrupts directly. Any interrupt-driven function should be implemented as a device driver module which should handle all interrupt-related functions. When it is necessary for a program to be synchronized to an interrupt-causing event, a driver can send a semaphore to a program (or the reverse) using OS-9's *signal* facilities.

It is important to understand that interrupt service routines are asynchronous and somewhat nebulous in that they are not distinct processes. They are in effect subroutines called by OS-9 when an interrupt occurs.

Therefore, all interrupt-driven device drivers have two basic parts: the "mainline" subroutines that execute as part of the calling process, and a separate interrupt service routine.

The two routines are asynchronous and therefore must use signals for communications and coordination.

The INIT initialization subroutine within the driver package should allocate static storage for the service routine, get the service routine address, and execute the F$IRQ system call to add it to the IRQ polling table.

When a device driver routine does something that will result in an interrupt, it should immediately execute a F$Sleep service request. This results in the process' deactivation. When the interrupt in question occurs, its service routine is executed after some random interval. It should then do the minimal amount of processing required, and send a "wakeup" signal to its associated process using the F$Send service request. It may also put some data in its static storage (I/O data and status) which is shared with its associated "sleeping" process.

Some time later, the device driver "mainline" routine is awakened by the signal, and can process the data or status returned by the interrupt service routine.

Using Standard I/O Paths

Programs should be written to use standard I/O paths wherever practical. Usually, this involves I/O calls that are intended to communicate to the user's terminal, or any other case where the OS-9 redirected I/O capability is desirable.

All three standard I/O paths will already be open when the program is entered (they are inherited from the parent process). Programs should *not* close these paths except under very special circumstances.

Standard I/O paths are always assigned path numbers zero, one, and two, as down below:

Path 0 - Standard Input. Analogous to the keyboard or other main data input source.

Path 1 - Standard Output. Analogous to the terminal display or other main data output destination.

Path 2 - Standard Error/Status. This path is provided so output messages which are not part of the actual program output can be kept separate. Many times paths 1 and 2 will be directed to the same device.

A Sample Program

The OS-9 **list** utility command program is shown on this and the next page as an example of assembly language programming.

```
Microware OS-9 Assembler 2.1    01/04/82 23:39:37         Page 001
LIST - File List Utility


     *****
     *  LIST UTILITY COMMAND
     *  Syntax: list <pathname>
     *  COPIES INPUT FROM SPECIFIED FILE TO STANDARD OUTPUT
0000 87CD0048           mod LSTEND,LSTNAM,PRGRM+OBJCT,
                            REENT+1,LSTENT,LSTMEM
000D 4C6973F4   LSTNAM   fcs   "List"

     * STATIC STORAGE OFFSETS
     *
00C8            BUFSIZ   equ   200        size of input buffer
0000                     ORG   0
0000            IPATH    rmb   1          input path number
0001            PRMPTR   rmb   2          parameter pointer
0003            BUFFER   rmb   BUFSIZ     allocate line buffer
00CB                     rmb   200        allocate stack
0193                     rmb   200        room for parameter list
025B            LSTMEM   EQU   .

0011 9F01       LSTENT   stx   PRMPTR     save parameter ptr
0013 8601                lda   #READ.     select read access mode
0015 103F84              os9   I$Open      open input file
0018 252E                bcs   LIST50     exit if error
001A 9700                sta   IPATH      save input path number
001C 9F01                stx   PRMPTR     save updated param ptr

001E 9600       LIST20   lda   IPATH      load input path number
0020 3043                leax  BUFFER,U   load buffer pointer
0022 10BE0C88            ldy   #BUFSIZ    maximum bytes to read
0026 103F8B              os9   I$ReadLn   read line of input
0029 2509                bcs   LIST30     exit if error
002B 8601                lda   #1         load std. out. path #
002D 103F8C              os9   I$WritLn   output line
0030 24EC                bcc   LIST20     Repeat if no error
0032 2014                bra   LIST50     exit if error
```

```
0034 C1D3       LIST30   cmpb  #E$EOF     at end of file?
0036 2610                bne   LIST50     branch if not
0038 9600                lda   IPATH      load input path number
003A 103F8F              os9   I$Close    close input path
003D 2509                bcs   LIST50     ..exit if error
003F 9E01                ldx   PRMPTR     restore parameter ptr
0041 A684                lda   0,X
0043 810D                cmpa  #$0D       End of parameter line?
0045 26CA                bne   LSTENT     ..no; list next file
0047 5F                  clrb
0048 103F06     LIST50   os9   F$Exit     ... terminate

004B 95BB58              emod             Module CRC

004E            LSTEND   EQU   *
```

Chapter 9. Adapting OS-9 to a New System

Thanks to OS-9's modular structure, it is easily portable to almost any 6809-based computer, and in fact it has been installed on an incredible variety of hardware. Usually only device driver and device descriptor modules need by rewritten or modified for the target system's specific hardware devices. The larger and more complex kernel and file manager modules almost never need adaptation.

One essential point is that you will need a functional OS-9 development system to use during installation of OS-9 on a new target system. Although it is possible to use a non-OS-9 system, or if you are truly masochistic, the target system itself, lack of facilities to generate and test memory modules and create system disks can make an otherwise straightforward job a time-consuming headache that is seldom less costly than a commercial OS-9-equipped computer. Over a dozen manufacturers offer OS-9 based development systems in all price ranges with an excellent selection of time-saving options such as hard disks, line printers. PROM programmers, etc.

A group of OS-9 aficionados has over the years first written an alternative to OS-9 called NitrOS9, then also disassembled all the original Microware code and commented the source code. The software is available on the Internet. You can find source code to the Kernel, Shell, INIT, SYSGO, device driver and descriptor modules, and a selection of utility commands which can be useful when moving OS-9 to a new target system.

Adapting OS-9 to Disk-based Systems

Usually, most of the work in moving OS-9 to a disk-based target system is writing a device driver module for the target system's disk controller. Part of this task involves producing a subset of the driver (mostly disk read functions) for use as a bootstrap module.

If terminal and/or parallel I/O for terminals, printers, etc., will use ACIA and/or PIA-type devices, the standard ACIA and PIA device driver modules may be used, or device drivers of your own design may be used in place of or in addition to these standard modules Device descriptor modules may also require adaptation to match device addresses and initialization required by the target system.

A CLOCK module may be adapted from a standard version, or a new one may be created. All other component modules, such as IOMAN, RBFMAN, SCFMAN, Shell, and utilities seldom require modification.

Using OS-9 in ROM-based Systems

One of OS-9's major features is its ability to reside in ROM memory and work effectively with ROMed applications programs written in assembler or high-level languages such as Basic09, Pascal, and C.

All the component modules of OS-9 (including all commands and utilities) are directly ROMable without modification. In some cases, particularly when the target system is to automatically execute an application program upon system start-up, it may be necessary to reassemble the two modules used during system startup, INIT and SYSGO.

The first step in designing a ROM-based system is to select which OS-9 modules to include in ROM. The following checklist is designed to help you do so:

a. Include OS9P1, OS9P2, SYSGO, and INIT. These modules are required in any OS-9 system.

b. If the target system is perform any I/O or interrupt functions include IOMAN.

c. If the target system is to perform I/O to character-oriented I/O devices using ACIAs, PIAs, etc., include SCFMAN, required device drivers (such as ACIA and PIA, and/or your own), and device descriptors as needed (such as TERM, T1, P, and/or your own). If device addresses and/or initialization functions need to be changed, the device descriptor modules must be modified before being ROMed.

d. If the target system is to perform disk I/O, include RBFMAN, and appropriate disk driver and device descriptor modules. As in (c) above, change device addresses and initialization if needed. If RBFMAN *will not* be included, the INIT and SYSGO modules *must* be altered to remove references to disk files.

e. If the target system requires multiprogramming, time-of-day, or other time-related functions, include a CLOCK module for the target system's real-time clock. Also consider how the clock is to be started,. You may want to ROM the **setime** command, or have SYSGO start the clock.

f. It the target system will receive commands manually, or if any application program uses `Shell` functions, include the SHELL and SYSGO modules, otherwise include a modified SYSGO module which calls your application program instead of `Shell`.

Adapting the Initialization Module

INIT is a module that contains system startup parameters. It *must* be in ROM in any OS-9 system (it usually resides in the same ROM as the kernel). It is a non-executable module named "INIT" and has type "system" (code $C). It is scanned once during the system startup. It begins with the standard header followed by:

MODULE OFFSET	
$9,$A,$B	This location contains an upper limit RAM memory address used to override OS-9's automatic end-of-RAM search so that memory may be reserved for I/O device addresses or other special purposes.
$C	Number of entries to create in the IRQ polling table. One entry is required for each interrupt- generating device control register.
$D	Number of entries to create in the system device table. One entry is required for each device in the system.
$E,$F	Offset to a string which is the name of the first module to be executed after startup, usually "SYSGO". There must always be a startup module.
$10,$11	Offset to the default directory name string (normally /D0). This device is assumed when device names are omitted from pathlists. If the system will not use disks (e.g., RBFMAN will not be used) this offset *must*be zero.
$12,$13	Offset to the initial standard path string (typically /TERM). This path is opened as the standard paths for the initial startup module. This offset *must* contain zero if there is none.
$14,$15	Offset to bootstrap module name string. If OS-9 does not find IOMAN in ROM during the start-up module search, it will execute the bootstrap module named to load additional modules from a file on a mass-storage device.
$16 to N	All name strings referred to above go here. Each must have the sign bit (bit 7) of the last character set.

Adapting the SYSGO Module

SYSGO is a program which is the first process started after the system start-up sequence. Its function is threefold:

- It does additional high-level system initialization, for example, disk system SYSGO call the `shell` to process the `Startup` shell procedure file.
- It starts the first "user" process.
- It thereafter remains in a "wait" state as insurance against all user processes terminating, thus leaving the system halted. If this happens. SYSGO can restart the first user program.

The standard SYSGO module for disk systems cannot be used on non-disk based systems unless it is modified to:

1. Remove initialization of the working execution directory.
2. Remove processing of the `Startup` procedure file.
3. Possibly change the name of the first user program from `Shell` to the name of a applications program. Here are some example name strings:

fcs /userpqm/	(object code module "userpgm")
fcs /RunB userpgm/	(compiled Basie09 program using RunB run-time-only system)
fcs /Basic09 userpgm/	(compiled Basic09 program using Basic09)

Chapter 10. OS-9 Service Request Descriptions

System calls are used to communicate between the OS-9 operating system and assembly-language-level programs. There are three general categories:

1. User mode function requests
2. System mode function requests
3. I/O requests

System mode function requests are privileged and may be executed only while OS-9 is in the system state (when it is processinq another service request, executing a file manager, device drivers, etc.). They are included in this manual primarily for the benefit of those programmers who will be writing device drivers and other system-level applications.

The system calls are performed by loading the MPU registers with the appropriate parameters (if any), and executing a SWI2 instruction immediately followed by a constant byte which is the request code. Parameters (if any) will be returned in the MPU registers after OS-9 has processed the service request. A standard convention for reporting errors is used in all system calls; if an error occurred, the "C bit" of the condition code register will be set and accumulator B will contain the appropriate error code. This permits a BCS or BCC instruction immediately following the system call to branch on error/no error.

Here is an example system call for the I$Close service request:

```
LDA PATHNUM
SWI2
FCB $8B
BCS ERROR
```

Using the assembler's "OS9" directive simplifies the call:

```
LDA PATHNUM
OS9 I$Close
BCS ERROR
```

The I/O service requests are simpler to use than in many other operating systems because the calling program does not have to allocate and set up "file control blocks", "sector buffers", etc. Instead OS-9 will return a one byte path number when a path to a file/device is opened or created; then this path number may be used in subsequent I/O requests to identify the file/device until the path is closed. OS-9 internally allocates and maintains its own data structures and users never have to deal with them: in fact attempts to do so are memory violations.

All system calls have a mnemonic name that starts with "F$" for system functions, or "I$" for I/O related requests. These are defined in the assembler-input equate file called `OS9Defs`.

In the service request descriptions which follow, registers not explicitly specified as input or output parameters are not altered. Strings passed as parameters are normally terminated by having bit seven of the last character set, a space character, or an end of line character.

F$AllBit - Set bits in an allocation bit map

ASSEMBLER CALL: OS9 F$AllBit

MACHINE CODE: 103F 13

INPUT:	(X) = Base address of allocation bit map. (D) = Bit number of first bit to set. (Y) = Bit count (number of bits to set)
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request sets bits in the allocation bit map specified by the X register.

Bit numbers range from 0..N-1, where N is the number of bits in the allocation bit map.

F$Chain - Load and execute a new primary module

ASSEMBLER CALL:	OS9 F$Chain
MACHINE CODE:	103F 05
INPUT:	(X) = Address of module name or file name (Y) = Parameter area size (256 byte pages) (U) = Beginning address of parameter area (A) = Lanquage / type code (B) = Optional data area size (256 byte pages)
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call is similar to F$Fork, but it does not create a new process. It effectively "resets" the calling process' program and data memory areas and begins execution of a new primary module. Open paths are not closed or otherwise affected.

This system call is used when it is necessary to execute an entirely new program, but without the overhead of creating a new process. It is functionally similar to a F$Fork followed by an F$Exit, but with less processing overhead.

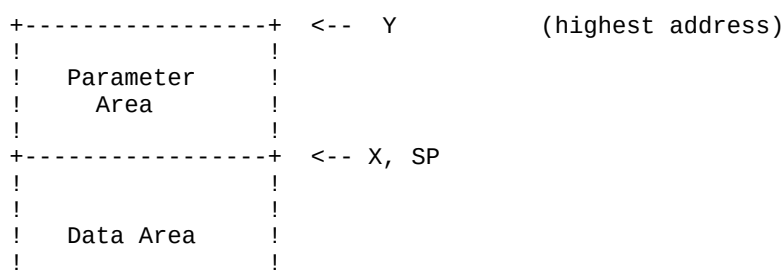
The sequence of operations taken by F$Chain is as follows:

1. The system parses the name string ot the new proces' "primary module" - the program that will initially be executed. Then the system module directory is searched to see if a module with the same name and type / language is already in memory. If so it is linked to. If not, the name string is used as the pathlist of a file which is to be loaded into memory. Then the first module in this file is linked to (several modules may have been loaded from a single file).

2. The process' old primary module is *unlinked*.

3. The data memory area is reconfigured to the size specified in the new primary module's header.

The diagram below shows how F$Chain sets up the data memory area and registers for the new module.



```
 !                !
 +----------------+
 !   Direct Page  !
 +----------------+  <-- U, DP       (lowest address)

 D = parameter area size
PC = module entry point abs. address
CC = F=0, I=0, others undefined
```

Y (top of memory pointer) and U (bottom of memory pointer) will always have a values at 256-byte page boundaries. If the parent does not specify a parameter area, Y, X, and SP will be the same, and D will equal zero. The minimum overall data area size is one page (256 bytes).

> **Warning**
>
> The hardware stack pointer (SP) should be located somewhere in the direct page before the F$Chain service request is executed to prevent a “suicide attempt” error or an actual suicide (system crash). This will prevent a suicide from occurring in case the new module requires a smaller data area than what is currently being used. You should allow approximately 200 bytes of stack space for execution of the F$Chain service request and other system “overhead”.

For more information, please see the F$Fork service request description.

F$CmpNam - Compare two names

ASSEMBLER CALL:	OS9 F$CmpNam
MACHINE CODE:	103F 11
INPUT:	(X) = Address of first name. (B) = Length of first name. (Y) = Address of second name.
OUTPUT:	(CC) = C bit clear if the strings match.

Given the address and length of a string, and the address of a second string, compares them and indicates whether they match. Typically used in conjunction with “parsename”.

The second name must bave the sign bit (bit 7) of the last character set.

F$CRC - Compute CRC

ASSEMBLER CALL:	OS9 F$CRC
MACHINE CODE:	103F 17
INPUT:	(X) = Starting byte address. (Y) = Byte count. (U) = Address of 3 byte CRC accumulator.
OUTPUT:	CRC accumulator is updated.
ERROR OUTPUT:	None.

This service request calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators. The CRC is calculated starting at the source address over “byte count” bytes, it is not necessary to cover an entire module in one call, since the CRC may be “accumulated” over several calls. The CRC accu-

mulator can be any three byte memory location and must be initialized to $FFFFFF before the first F$CRC call.

The last three bytes in the module (where the three CRC bytes will be stored) are not included in the CRC generation.

The polynomial is $800063. If you perform the CRC over the module minus the stored CRC then you must exclusive-or the result with $FFFFFF to compare directly with the stored CRC. If you perform CRC over the module including the last three bytes then the result must be $800FE3.

Example C code of CRC algorithm (with 32-bit longs):

```
unsigned long compute_crc()
    unsigned long crc;
    unsigned char *octets;
    int len;
{
    int i;

    while (len--) {
        crc ^= (*octets++) << 16;
        for (i = 0; i < 8; i++) {
            crc <<= 1;
            if (crc & 0x1000000L)
                crc ^= 0x800063L;
        }
    }
    return crc & 0xffffffL;
}
```

F$DelBit - Deallocate in a bit map

ASSEMBLER CALL:	OS9 F$DelBit
MACHINE CODE:	103F 14
INPUT:	(X) = Base address of an allocation bit map. (D) = Bit number of first bit to clear. (Y) = Bit count (number of bits to clear).
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request is used to clear bits in the allocation bit map pointed to by X.

Bit numbers range from 0..N-1, where N is the number of bits in the allocation bit map.

F$Exit - Terminate the calling process

ASSEMBLER CALL:	OS9 F$Exit
MACHINE CODE:	103F 06
INPUT:	(B) = Status code to be returned to the parent process
OUTPUT:	Process is terminated.

This call kills the calling process and is the only means by which a process can terminate itself. Its data memory area is deallocated, and its primary module is UNLINKed. All open paths are automatically closed.

The death of the process can be detected by the parent executing a F$Wait call, which returns to the parent the status byte passed by the child in its F$Exit call. The status byte can be an OS-9 error code the terminating process wishes to pass back to its parent process (the shell assumes this), or can be used to pass a user-defined status value. Processes to be called directly by the shell should only return an OS-9 error code or zero if no error occurred.

F$Fork - Create a new process

ASSEMBLER CALL:	OS9 F$Fork
MACHINE CODE:	103F 03
INPUT:	(X) = Address of module name or file name. (Y) = Parameter area size. (U) = Beginning address of the parameter area. (A) = Language / Type code. (B) = Optional data area size (pages).
OUTPUT:	(X) = Updated past the name string. (A) = New process ID number.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call creates a new process which becomes a “child” of the caller, and sets up the new process’ memory and MPU registers.

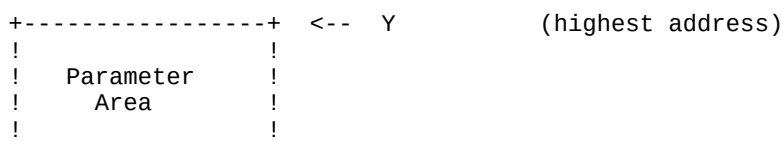
The system parses the name string of the new process’ “primary module” - the program that will initially be executed. Then the system module directory is searched to see if the program is already in memory. If so, the module is linked to and executed. If not, the name string is used as the pathlist of the file which is to loaded into memory. Then the first module in this file is linked to and executed (several modules may have been loaded from a single file).

The primary module’s module header is used to determine the process’ initial data area size. OS-9 then attempts to allocate a contiguous RAM area equal to the required data storage size, (includes the parameter passing area, which is copied from the parent process’ data area). The new process’ registers are set up as shown in the diagram on the next page. The execution offset given in the module header is used to set the PC to the module’s entry point.

When the `shell` processes a command line it passes a string in the parameter area which is a copy of the parameter part (if any) of the command line. It also inserts an end-of-line character at the end of the parameter string to simplify string-oriented processing. The X register will point to the beginning of the parameter string. If the command line included the optional memory size specification (#n or #nK), the `shell` will pass that size as the requested memory size when executing the F$Fork.

If any of the above operations are unsuccessful, the F$Fork is aborted and the caller is returned an error.

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



```
  +-----------------+  <-- X, SP
  !                 !
  !                 !
  !   Data Area     !
  !                 !
  !                 !
  +-----------------+
  !   Direct Page   !
  +-----------------+  <-- U, DP       (lowest address)

   D = parameter area size
  PC = module entry point abs. address
  CC = F=0, I=0, others undefined
```

Y (top of memory pointer) and U (bottom of memory pointer) will always have a values at 256-byte page boundaries. If the parent does not specify a parameter area, Y, X, and SP will be the same, and D will equal zero. The minimum overall data area size is one page (256 bytes). `Shell` will always pass at least an end of line character in the parameter area.

NOTE: Both the child and parent process will execute concurrently. If the parent executes a F$Wait call immediately after the fork, it will wait until the child dies before it resumes execution. Caution should be exercised when recursively calling a program that uses the F$Fork service request since another child may be created with each "incarnation". This will continue until the process table becomes full.

F$ICPT - Set up a signal intercept trap

ASSEMBLER CALL:	OS9 F$ICPT
MACHINE CODE:	103F 09
INPUT:	(X) = Address of the intercept routine. (U) = Address of the intercept routine local storage.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call tells OS-9 to set a signal intercept trap, where X contains the address of the signal handler routine, and U contains the base address of the routine's storage area. After a signal trap has been set, whenever the process receives a signal, its intercept routine will be executed. A signal will abort any process which has not used the F$ICPT service request to set a signal trap, and its termination status (B register) will be the signal code. Many interactive programs will set up an intercept routine to handle keyboard abort (**controlQ**), and keyboard interrupt (**controlC**).

The intercept routine is entered asynchronously because a signal may be sent at any time (it is like an interrupt) and is passed the following:

U = Address of intercept routine local storage.

B = Signal code.

NOTE: The value of DP may not be the same as it was when the F$ICPT call was made.

Whenever a signal is received. OS-9 will pass the signal code and the base address of its data area (which was defined by a F$ICPT service request) to the signal intercept routine. The base address of the data area is selected by the user and is typically a pointer to the process' data area.

The intercept routine is activated when a signal is received, then it takes some action based upon the value of the signal code such as setting a flag in the process' data

area. After the signal has been processed, the handler routine should terminate with an RTI instruction.

F$ID - Get process ID / user ID

ASSEMBLER CALL:	OS9 F$ID
MACHINE CODE:	103F 0C
INPUT:	None
OUTPUT:	(A) = Process ID. (Y) = User ID.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Returns the caller's process ID number, which is a byte value in the range of 1 to 255, and the user ID which is a integer in the range 0 to 65535. The process ID is assigned by OS-9 and is unique to the process. The user ID is defined in the system password file, and is used by the file security system and a few other functions. Several processes can have the same user ID.

F$LINK - Link to memory module

ASSEMBLER CALL:	OS9 F$LINK
MACHINE CODE:	103F 00
INPUT:	(X) = Address of the module name string. (A) = Module type / language byte.
OUTPUT:	(X) = Advanced past the module name. (Y) = Module entry point absolute address. (U) = Module header absolute address. (A) = Module type / language. (B) = Module attributes / revision level.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call causes OS-9 to search the module directory for a module having a name, language and type as given in the parameters. If found, the address of the module's header is returned in U, and the absolute address of the module's execution entry point is returned in Y (as a convenience: this and other information can be obtained from the module header). The module's link count' is incremented whenever a F$LINK references its name, thus keeping track of how many processes are using the module. If the module requested has an attribute byte indicating it is not sharable (meaning it is not reentrant) only one process may link to it at a time.

Possible errors:

(A) Module not found.

(B) Module busy (not sharable and in use).

(C) Incorrect or defective module header.

F$LOAD - Load module(s) from a file

ASSEMBLER CALL:	OS9 F$LOAD
MACHINE CODE:	103F 01

INPUT:	(X) = Address of pathlist (file name) (A) = Language / type (0 = any language / type)
OUTPUT:	(X) = Advanced past pathlist (Y) = Primary module entry point address (U) = Address of module header (A; - Language / type (B) = Attributes / revision level
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Opens a file specified by the pathlist, reads one or more memory modules from the file into memory, then closes the file. All modules loaded are added to the system module directory, and the first module read is LINKed. The parameters returned are the same as the F$LINK call and apply only to the first module loaded.

In order to be loaded, the file must have the "execute" permission and contain a module or modules that have a proper module header. The file will be loaded from the working execution directory unless a complete pathlist is given.

Possible errors: module directory full; memory full; plus errors that occur on I$Open, I$Read, I$Close and F$LINK system calls.

F$Mem - Resize data memory area

ASSEMBLER CALL:	OS9 F$Mem
MACHINE CODE:	103F 07
INPUT:	(D) = Desired new memory area size in bytes
OUTPUT:	(Y) = Address of new memory area upper bound (D) = Actual new memory area size in bytes
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Used to expand or contract the process' data memory area. The new size requested is rounded up to the next 256-byte page boundary. Additional memory is allocated contiguously upward (towards higher addresses), or deallocated downward from the old highest address. If D = 0, then the current upper bound and size will be returned.

This request can never return all of a process' memory, or the page in which its SP register points to.

In Level One systems, the request may return an error upon an expansion request even though adequate free memory exists. This is because the data area is always made contiguous, and memory requests by other processes may fragment free memory into smaller, scattered blocks that are not adjacent to the caller's present data area. Level Two systems do not have this restriction because of the availability of hardware for memory relocation, and because each process has its own "address space".

F$PErr - Print error message

ASSEMBLER CALL:	OS9 F$PErr
MACHINE CODE:	103F 0F
INPUT:	(A) = Output path number. (B) = Error code.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This is the system's error reporting utility. It writes an error message to the output path specified. Most OS-9 systems will display:

ERROR #<decimal number>

by default. The error reporting routine is vectored and can be replaced with a more elaborate reporting module. To replace this routine use the F$SSVC service request.

F$PrsNam - Parse a path name

ASSEMBLER CALL:	OS9 F$PrsNam
MACHINE CODE:	103F 10
INPUT:	(X) = Address of the pathlist
OUTPUT:	(X) = Updated past the optional "/" (Y) = Address of the last character of the name + 1. (B) = Length of the name
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code. (X) = Updated past space characters.

Parses the input text string for a legal OS-9 name. The name is terminated by any character that is not a legal component character. This system call is useful for processing pathlist arguments passed to new processes. Also if X was at the end of a pathlist, a bad name error will be returned and X will be moved past any space characters so that the next pathlist in a command line may be parsed.

Note that this system call processes only one name, so several calls may be needed to process a pathlist that has more than one name.

BEFORE F$PrsNam CALL:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---
! / ! D ! 0 ! / ! F ! I ! L ! E !   !   !   !   !
+---+---+---+---+---+---+---+---+---+---+---+---+---+---
  ^
  X
```

AFTER THE F$PrsNam CALL:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---
! / ! D ! 0 ! / ! F ! I ! L ! E !   !   !   !   !
+---+---+---+---+---+---+---+---+---+---+---+---+---+---
      ^       ^
      X       Y       (B) = 2
```

F$SchBit - Search bit map for a free area

ASSEMBLER CALL:	OS9 F$SchBit
MACHINE CODE:	103F 12
INPUT:	(X) = Beginning address of a bit map. (D) = Beginning bit number. (Y) = Bit count (free bit block size). (U) = End of bit map address.
OUTPUT:	(D) = Beginning bit number. (Y) = Bit count.

This system mode service request searches the specified allocation bit map starting

at the "beginning bit number" for a free block (cleared bits) of the required length.

If no block of the specified size exists, it returns with the carry set, beginning bit number and size of the largest block.

F$Send - Send a signal to another process

ASSEMBLER CALL:	OS9 F$Send
MACHINE CODE:	103F 08
INPUT:	(A) = Receiver's process ID number. (B) = Signal code.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call sends a "signal" to the process specified. The signal code is a single byte value of 1 - 255.

If the signal's destination process is sleeping or waiting, it will be activated so that it may process the signal. The signal processing routine (intercept) will be executed if a signal trap was set up (see F$ICPT), otherwise the signal will abort the destination process, and the signal code becomes the exit status (see F$Wait). An exception is the WAKEUP signal, which activates a sleeping process but does not cause the signal intercept routine to be executed.

Some of the signal codes have meanings defined by convention:

0 = System Abort (cannot be intercepted)
1 = Wake Up Process
2 = Keyboard Abort
3 = Keyboard Interrupt
4-255 = user defined

If an attempt is made to send a signal to a process that has an unprocessed, previous signal pending, the current "send" request will be cancelled and an error will be returned. An attempt can be made to re-send the signal later. It is good practice to issue a "sleep" call for a few ticks before a retry to avoid wasting MPU time.

For related information see the F$ICPT, F$Wait and F$Sleep service request descriptions.

F$Sleep - Put calling process to sleep

ASSEMBLER CALL:	OS9 F$Sleep
MACHINE CODE:	103F 0A
INPUT:	(X) = Sleep time in ticks (0 = indefinitely)
OUTPUT:	(X) = Decremented by the number of ticks that the process was asleep.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This call deactivates the calling process for a specified time, or indefinitely if X = 0. If X = 1, the effect is to have the caller give up its current time slice. The process will be activated before the full time interval if a signal is received, therefore sleeping indefinitely is a good way to wait for a signal or interrupt without wasting CPU time.

The duration of a "tick" is system dependent but is most commonly 100 milliseconds.

Due to the fact that it is not known when the F$Sleep request was made during the current tick, F$Sleep can not be used for precise timing. A sleep of one tick is effectively a "give up remaining time slice" request; the process is immediately inserted into the active process queue and will resume execution when it reaches the front of the queue. A sleep of two or more ticks causes the process to be inserted into the active process queue after N-1 ticks occur and will resume execution when it reaches the front of the queue.

F$SPrior - Set process priority

ASSEMBLER CALL:	OS9 F$SPrior
MACHINE CODE:	103F 0D
INPUT:	(A) = Process ID number. (B) = Priority: 0 = lowest 255 - highest
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Changes the process' priority to the new value given. $FF is the highest possible priority, $00 is the lowest. A process can change another process' priority only if it has the same user ID.

F$SSVC - Install function request

ASSEMBLER CALL:	OS9 F$SSVC
MACHINE CODE:	103F 32
INPUT:	(Y) = Address of service request initialization table.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request is used to add a new function request to OS-9's user and privileged system service request tables, or to replace an old one. The Y register passes the address of a table which contains the function codes and offsets to the corresponding service request handler routines. This table has the following format:

```
OFFSET

          +----------------------+
 $00      !     Function Code    !  <--- First entry
          +----------------------+
 $01      ! Offset From Byte 3   !
          +--                  --+
 $02      !  To Function Handler !
          +----------------------+
 $03      !     Function Code    !  <--- Second entry
          +----------------------+
 $04      ! Offset From Byte 6   !
          +--                  --+
 $05      !  To Function Handler !
          +----------------------+
          !                      !   <--- Third entry etc.
          !     MORE ENTRIES     !
          !                      !
```

```
          !                      !
          +----------------------+
          !         $80          !   <--- End of table mark
          +----------------------+
```

NOTE: If the sign bit of the function code is set, only the system table will be updated. Otherwise both the system and user tables will be updated. Privileged system service requests may be called only while executing a system routine.

The service request handler routine should process the service request and return from subroutine with an RTS instruction. They may alter all MPU registers (except for SP). The U register will pass the address of the register stack to the service request handler as shown in the following diagram:

```
                                OFFSET   OS9Defs
                                         MNEMONIC
        +------+
U --->  !  CC  !                  $0       R$CC
        +------+                  $1       R$D
        !  A   !                  $1       R$A
        +------+
        !  B   !                  $2       R$B
        +------+
        !  DP  !                  $3       R$DP
        +------+------+
        !      X      !           $4       R$X
        +-------------+
        !      Y      !           $6       R$Y
        +-------------+
        !      U      !           $8       R$U
        +-------------+
        !      PC     !           $A       R$PC
        +-------------+
```

Function request codes are broken into the two categories as shown below:

$00 - $27	User mode service request codes.
$29 - $34	Privileged system mode service request codes. When installing these service request, the sign bit should be set if it is to be placed into the system table only.

NOTE: These categories are defined by convention and not enforced by OS9.

Codes $25..$27, and $70..$7F will not be used by the operating system and are free for user definition.

F$SSWI - Set SWI vector

ASSEMBLER CALL:	OS9 F$SSWI
MACHINE CODE:	103F 0E
INPUT:	(A) = SWI type code. (X) = Address of user SWI service routine.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Sets up the interrupt vectors for SWI, SWI2 and SWI3 instructions. Each process has its own local vectors. Each F$SSWI call sets up one type of vector according to the

code number passed in A.

1 = SWI
2 = SWI2
3 = SWI3

When a process is created, all three vectors are initialized with the address of the OS-9 service call processor.

Warning

Software built for OS-9 uses SWI2 to call OS-9. If you reset this vector these programs will not work. If you change all three vectors, you will not be able to call OS-9 at all.

F$STime - Set system date and time

ASSEMBLER CALL:	OS9 F$STime
MACHINE CODE:	103F 16
INPUT:	(X) = Address of time packet (see below)
OUTPUT:	Time/date is set.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This service request is used to set the current system date/time and start the system real-time clock. The date and time are passed in a time packet as follows:

OFFSET	VALUE
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

F$Time - Get system date and time

ASSEMBLER CALL:	OS9 F$Time
MACHINE CODE:	103F 15
INPUT:	(X) = Address of place to store the time packet.
OUTPUT:	Time packet (see below).
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This returns the current system date and time in the form of a six byte packet (in binary). The packet is copied to the address passed in X. The packet looks like:

OFFSET	VALUE

OFFSET	VALUE
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

F$Unlink - Unlink a module

ASSEMBLER CALL:	OS9 F$Unlink
MACHINE CODE:	103F 02
INPUT:	(U) = Address of the module header.
OUTPUT:	None
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Tells OS-9 that the module is no longer needed by the calling process. The module's link count is decremented, and the module is destroyed and its memory deallocated when the link count equals zero. The module will not be destroyed if in use by any other process(es) because its link count will be non-zero. In Level Two systems, the module is usually switched out of the process' address space.

Device driver modules in use or certain system modules cannot he unlinked. ROMed modules can be unlinked but cannot be deleted from the module directory.

F$Wait - Wait for child process to die

ASSEMBLER CALL:	OS9 F$Wait
MACHINE CODE:	103F 04
INPUT:	None
OUTPUT:	(A) = Deceased child process' process ID (B) = Child process' exit status code
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

The calling process is deactivated until a child process terminates by executing an F$Exit system call, or by receiving a signal. The child's ID number and exit status is returned to the parent. If the child died due to a signal, the exit status byte (B register) is the signal code.

If the caller has several children, the caller is activated when the first one dies, so one F$Wait system call is required to detect termination of each child.

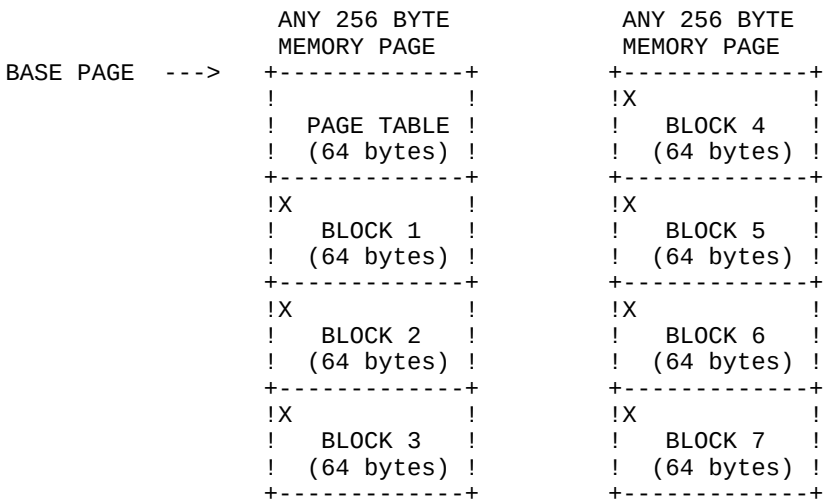
If a child died before the F$Wait call, the caller is reactivated almost immediately. F$Wait will return an error if the caller has no children.

See the F$Exit description for more related information.

F$All64 - Allocate a 64 byte memory block

ASSEMBLER CALL:	OS9 F$All64
MACHINE CODE:	103F 30
INPUT:	(X) = Base address of page table (zero if the page table has not yet been allocated).
OUTPUT:	(A) = Block number (X) = Base address of page table (Y) = Address of block.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request is used to dynamically allocate 64 byte blocks of memory by splitting whole pages (256 byte) into tour sections. The first 64 bytes of the base page are used as a "page table", which contains the MSB of all pages in the memory structure. Passing a value of zero in the X register will cause the F$All64 service request to allocate a new base page and the first 64 byte memory block. Whenever a new page is needed, an F$SRqMem service request will automatically be executed. The first byte of each block contains the block number; routines using this service request should not alter it. Below is a diagram to show how 7 blocks might be allocated:



Note: This is a privileged system mode service request.

F$AProc - Insert process in active process queue

ASSEMBLER CALL:	OS9 F$AProc
MACHINE CODE:	103F 2C
INPUT:	(X) = Address of process descriptor.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request inserts a process into the active process queue so that it may be scheduled for execution.

All processes already in the active process queue are aged, and the age of the specified

process is set to its priority. If the process is in system state, it is inserted after any other process's also in system state, but before any process in user state. If the process is in user state, it is inserted according to its age.

Note: This is a privileged system mode service request.

F$Find64 - Find a 64 byte memory block

ASSEMBLER CALL:	OS9 F$Find64
MACHINE CODE:	103F 2F
INPUT:	(X) = Address of base page. (A) = Block number.
OUTPUT:	(Y) = Address of block.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request will return the address of a 64 byte memory block as described in the F$All64 service request. OS-9 used this service request to find process descriptors and path descriptors when given their number.

Block numbers range from 1..N

Note: This is a privileged system mode service request.

F$IODel - Delete I/O device from system

ASSEMBLER CALL:	OS9 F$IODel
MACHINE CODE:	103F 33
INPUT:	(X) = Address of an I/O module, (see description).
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request is used to determine whether or not an I/O module is being used. The X register passes the address of a device descriptor module, device driver module, or file manager module. The address is used to search the device table, and if found the use count is checked to see if it is zero. If it is not zero, an error condition is returned.

This service request is used primarily by IOMAN and may be of limited or no use for other applications.

Note: This is a privileged system mode service request.

F$IOQU - Enter I/O queue

ASSEMBLER CALL:	OS9 F$IOQU
MACHINE CODE:	103F 2B
INPUT:	(A) = Process Number.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request links the calling process into the I/O queue of the specified process and performs an untimed sleep. It is assumed that routines associated with the specified process will send a wakeup signal to the calling process.

Note: This is a privileged system mode service request.

F$IRQ - Add or remove device from IRQ table

ASSEMBLER CALL:	OS9 F$IRQ
MACHINE CODE:	103F 2A
INPUT:	(X) = Zero to remove device from table, or the address of a packet as defined below to add a device to the IRQ polling table: [x] = flip byte [X+1] = mask byte [X+2] = priority (U) = Address of service routine's static storage area. (Y) = Device IRQ service routine address. (D) = Address of the device status register.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This service request is used to add a device to or remove a device from the IRQ polling table. To remove a device from the table the input should be (X)=0, (U)= Addr of service routine's static storage. This service request is primarily used by device driver routines. See the text of this manual for a complete discussion of the interrupt polling system.

PACKET DEFINITIONS:

Flip Byte	This byte selects whether the bits in the device status register are active when set or active when cleared. A set bit(s) identifies the active bit(s).
Mask Byte	This byte selects one or more bits within the device status register that are interrupt request flag(s). A set bit identifies an active bit(s)
Priority	The device priority number: 0 = lowest 255 = highest

Note: This is a privileged system mode service request.

F$NProc - Start next process

ASSEMBLER CALL:	OS9 F$NProc
MACHINE CODE:	103F 2D
INPUT:	None.
OUTPUT:	Control does not return to caller.

This system mode service request takes the next process out of the Active Process Queue and initiates its execution. If there is no process in the queue, OS-9 waits for an interrupt, and then checks the active process queue again.

Note: This is a privileged system mode service request.

F$Ret64 - Deallocate a 64 byte memory block

ASSEMBLER CALL:	OS9 F$Ret64
MACHINE CODE:	103F 31
INPUT:	(X) = Address of the base page. (A) = Block number.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request deallocates a 64 byte block of memory as described in the F$All64 service request.

Note: This is a privileged system mode service request.

F$SRqMem - System memory request

ASSEMBLER CALL:	OS9 F$SRqMem
MACHINE CODE:	103F 28
INPUT:	(D) = Byte count.
OUTPUT:	(U) = Beginning address of memory area.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request allocates a block of memory from the top of available RAM of the specified size. The size requested is rounded to the next 256 byte page boundary.

Note: This is a privileged system mode service request.

F$SRtMem - Return System Memory

ASSEMBLER CALL:	OS9 F$SRtMem
MACHINE CODE:	103F 29
INPUT:	(U) = Beginning address of memory to return. (D) = Number of bytes to return.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request is used to deallocate a block of contigous 256 byte pages. The U register must point to an even page boundary.

Note: This is a privileged system mode service request.

F$VModul - Verify module

ASSEMBLER CALL:	OS9 F$VModul
MACHINE CODE:	103F 2E
INPUT:	(X) = Address of module to verify
OUTPUT:	(U) = Address of module directory entry
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system mode service request checks the module header parity and CRC bytes of an OS-9 module. If these values are valid, then the module directory is searched for a module with the same name. If a module with the same name exists, the one with the highest revision level is retained in the module directory. Ties are broken in favor of the established module.

Note: This is a privileged system mode service request.

I$Attach - Attach a new device to the system

ASSEMBLER CALL:	OS9 I$Attach
MACHINE CODE:	103F 80
INPUT:	(X) = Address of device name string (A) = Access mode.
OUTPUT:	(U) = Address of device table entry
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This service request is used to attach a new device to the system, or verify that it is already attached. The device's name string is used to search the system module directory to see if a device descriptor module with the same name is in memory (this is the name the device will be known by). The descriptor module will contain the name of the device's file manager, device driver and other related information. It it is found and the device is not already attached, OS-9 will link to its file manager and device driver, and then place their address' in a new device table entry. Any permanent stor-

age needed by the device driver is allocated, and the driver's initialization routine is called (which usually initializes the hardware).

If the device has already been attached, it will not be reinitialized.

An I$Attach system call is not required to perform routine I/O. It does *not* "reserve" the device in question - it just prepares it for subsequent use by any process. Most devices are automatically installed, so it is used mostly when devices are dynamically installed or to verify the existence of a device.

The access mode parameter specifies which subsequent read and/or write operations will be permitted as follows:

0 = Use device capabilities.

1 = Read only.

2 = Write only.

3 = Both read and write.

I$ChgDir - Change working directory

ASSEMBLER CALL:	OS9 I$ChgDir
MACHINE CODE:	103F 86
INPUT:	(X) = Address of the pathlist. (A) = Access mode.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Changes a process' working directory to another directory file specified by the pathlist. Depending on the access mode given, the current execution or the current data directory may be changed (but only one may be changed per call). The file specified must be a directory file, and the caller must have read permission for it (public read if not owned by the calling process).

ACCESS MODES

1 = Read

2 = Write

3 = Update (read or write)

4 = Execute

If the access mode is read, write, or update the current data directory is changed. If the access mode is execute, the current execution directory is changed.

I$Close - Close a path to a file/device

ASSEMBLER CALL:	OS9 I$Close
MACHINE CODE:	103F 8F
INPUT:	(A) = Path number.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Terminates the I/O path specified by the path number. I/O can no longer be performed to the file/device, unless another I$Open or I$Create call is used. Devices that are non-sharable become available to other requesting processes. All OS-9 inter-

nally managed buffers and descriptors are deallocated.

Note: Because the OS9 F$Exit service request automatically closes all open paths (except the standard I/O paths), it may not he necessary to close them individually with the OS9 I$Close service request.

Standard I/O paths are not typically closed except when it is desired to change the files/devices they correspond to.

I$Create - Create a path to a new file

ASSEMBLER CALL:	OS9 I$Create
MACHINE CODE:	103F 83
INPUT:	(X) = Address of the pathlist. (A) = Access mode. (B) = File attributes.
OUTPUT:	(X) = Updated past the pathlist (trailing blanks skipped) (A) = Path number.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Used to create a new file on a multifile mass storage device. The pathlist is parsed, and the new file name is entered in the specified (or default working) directory. The file is given the attributes passed in the B register, which has individual bits defined as follows:

bit 0 = read permit
bit 1 = write permit
bit 2 = execute permit
bit 3 = public read permit
bit 4 = public write permit
bit 5 - public execute permit
bit 6 = sharable file

The access mode parameter passed in register A must be either “WRITE” or “UPDATE”. This only affects the file until it is closed; it can be reopened later in any access mode allowed by the file attributes (see I$Open). Files open for “WRITE” may allow faster data transfer than “UPDATE”, which sometimes needs to preread setors. These access codes are defined as given below:

2 = Write only
3 = Update (read and write)

NOTE: If the execute bit (bit 2) is set, the file will be created in the working execution directory instead of the working data directory.

The path number returned by OS-9 is used to identify the file in subsequent I/O service requests until the file is closed.

No data storage is initially allocated for the file at the time it is created; this is done automatically by I$Write or explicitly by the I$SetStt call.

An error will occur if the file name already exists in the directory. I$Create calls that specify non-multiple file devices (such as printers, terminals, etc.) work correctly: the I$Create behaves the same as I$Open. Create cannot be used to make directory files (see I$MakDir).

I$Delete - Delete a file

ASSEMBLER CALL:	OS9 I$Delete
MACHINE CODE:	103F 87
INPUT:	(X) = Address of pathlist.
OUTPUT:	(X) = Updated past pathlist (trailing spaces skipped).
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This service request deletes the file specified by the pathlist. The file must have write permission attributes (public write if not the owner), and reside on a multifile mass storage device. Attempts to delete devices will result in an error.

I$Detach - Remove a device from the system

ASSEMBLER CALL:	OS9 I$Detach
MACHINE CODE:	103F 81
INPUT:	(U) = Address of the device table entry.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Removes a device from the system device table if not in use by any other process. The device driver's termination routine is called, then any permanent storage assigned to the driver is deallocated. The device driver and file manager modules associated with the device are unlinked (and may be destroyed if not in use by another process.

The I$Detach service request must be used to un-attach devices that were attached with the I$Attach service request. Both of these are used mainly by IOMAN and are of limited (or no use) to the typical user. SCFMAN also uses I$Attach/I$Detach to setup its second (echo) device.

I$Dup Duplicate a path

ASSEMBLER CALL:	OS9 I$Dup
MACHINE CODE:	103F 82
INPUT:	(A) = Path number of path to duplicate.
OUTPUT:	(A) = New path number.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Given the number of an existing path, returns another synonymous path number for the same file or device. `Shell` uses this service request when it redirects I/O. Service requests using either the old or new path numbers operate on the same file or device.

NOTE: This only increments the "use count" of a path descriptor and returns the synonymous path number. The path descriptor is not copied.

I$GetStt - Get file device status

ASSEMBLER CALL:	OS9 I$GetStt
MACHINE CODE:	103F 8D

INPUT:	(A) = Path number. (B) Status code. (Other registers depend upon status code)
OUTPUT:	(depends upon status code)
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system is a "wild card" call used to handle individual device parameters that:

a. are not uniform on all devices

b. are highly hardware dependent

c. need to be user-changable

The exact operation of this call depends on the device driver an file manager associated with the path. A typical use is to determine a terminal's parameters for backspace character, delete character, echo on/off, null padding, paging, etc. It is commonly used in conjunction with the I$SetStt service request which is used to set the device operating parameters. Below are presently defined function codes for I$GetStt:

MNEMONIC	CODE	FUNCTION
SS.Opt	0	Read the 32 byte option section of the path descriptor.
SS.Ready	1	Test for data ready on SCFMAN-type device.
SS.Size	2	Return current file size (on RBFMAN-type devices).
SS.Pos	5	Get current file position.
SS.EOF	6	Test for end of file.
SS.ScSiz	38	Width of screen in characters.

CODES 7-127 Reserved for future use.

CODES 128-255 These getstat codes and their parameter passing conventions are user definable (see the sections of this manual on writing device drivers). The function code and register stack are passed to the device driver.

Parameter Passing Conventions

The parameter passing conventions for each of these function codes are given below:

SS.Opt (code 0):	Read option section of the path descriptor.
INPUT:	(A) = Path number (B) = Function code 0 (X) = Address of place to put a 32 byte status packet.
OUTPUT:	Status packet.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This getstat function reads the option section of the path descriptor and copies it into the 32 byte area pointed to by the X register. It is typically used to determine the current settings for echo, auto line feed, etc. For a complete description of the status packet, please see the section of this manual on path descriptors.

SS.Ready (code 1):	Test for data available on SCFMAN supported devices.
INPUT:	(A) = Path number (B) = Function code 1
OUTPUT:	ENTRYTBL not supported.

SS.Size (code 2):	Get current file size (RBFMAN supported devices only)
INPUT:	(A) = Path number (B) = Function code 2
OUTPUT:	(X) = M.S. 16 bits of current file size. (U) = L.S. 16 bits of current file size.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

SS.Pos (code 5):	Get current file position (RBFMAN supported devices only)
INPUT:	(A) = Path number (B) = Function code 5
OUTPUT:	(X) = M.S. 16 bits of current file position. (U) = L.S. 16 bits of current file position.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

SS.EOF (code 6):	Test for end of file.
INPUT:	(A) = Path number (B) = Function code 6
OUTPUT:	ENTRYTBL not supported.

SS.ScSiz (code 38):	Return screen size for COCO (SCFMAN supported devices only)
INPUT:	(A) = Path number (B) = Function code 38
OUTPUT:	(X) = Width of screen in characters. Typically 32 or 80.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

I$MakDir - Make a new directory

ASSEMBLER CALL:	OS9 I$MakDir
MACHINE CODE:	103F 85
INPUT:	(X) = Address of pathlist. (B) = Directory attributes.
OUTPUT:	(X) = Updated path pathlist (trailing spaces skipped).
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

I$MakDir is the only way a new directory file can be created. It will create and initialize a new directory as specified by the pathlist. The new directory file contains no entries, except for an entry for itself (".") and its parent directory ("..")

The caller is made the owner of the directory. I$MakDir does not return a path number because directory files are not “opened” by this request (use I$Open to do so). The new directory will automatically have its “directory” bit set in the access permission attributes. The remaining attributes are specified by the byte passed in the B register, which has individual bits defined as follows:

bit 0 = read permit
bit 1 = write permit
bit 2 = execute permit
bit 3 = public read permit
bit 4 = public write permit

bit 5 - public execute permit
bit 6 = sharable file
bit 7 = (don't care)

I$Open - Open a path to a file or device

ASSEMBLER CALL:	OS9 I$Open
MACHINE CODE:	103F 84
INPUT:	(X) = Address of pathlist. (A) = Access mode (D S PE PW PR E W R)
OUTPUT:	(X) = Updated past pathlist (trailing spaces skipped). (A) = Path number.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Opens a path to an existing file or device as specified by the pathlist. A path number is returned which is used in subsequent service requests to identify the file.

The access mode parameter specifies which subsequent read and/or write operation are permitted as follows:

1 = read mode
2 = write mode
3 = update mode (both read and write)

Update mode can be slightly slower because pre-reading of sectors may be required for random access of bytes within sectors. The access mode must conform to the access permission attributes associated with the file or device (see I$Create). Only the owner may access a file unless the appropiate "public permit" bits are set.

Files can be opened by several processes (users) simultaneously. Devices have an attribute that specifies whether or not they are sharable on an individual basis.

NOTES:

If the execution bit is set in the access mode, OS-9 will begin searching for the file in the working execution directory (unless the pathlist begins with a slash).

The sharable bit (bit 6) in the access mode can not lock other users out of file in OS-9 Level I. It is present only for upward compatibility with OS-9 Level II.

Directory files may be read or written if the D bit (bit 7) is set in the access mode.

I$Read - Read data from a file or device

ASSEMBLER CALL:	OS9 I$Read
MACHINE CODE:	103F 89
INPUT:	(X) = Address to store data. (Y) = Number of bytes to read. (A) = Path number.
OUTPUT:	(Y) = Number of bytes actually read.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Reads a specified number of bytes from the path number given. The path must previously have been opened in READ or UPDATE mode. The data is returned exactly as read from the file/device without additional processing or editing such as backspace, line delete, end-of-file, etc.

After all data in a file has been read, the next I$Read service request will return an end of file error.

NOTES:

The keyboard abort, keyboard interrupt, and end-of-file characters may be filtered out of the input data on SCFMAN-type devices unless the coresponding entries in the path descriptor have been set to zero. It may be desirable to modify the device descriptor so that these values in the path descriptor are initialized to zero when the path is opened.

The number of bytes requested will be read unless:

A. An end-of-file occurs
B. An end-of-record occurs (SCFMAN only)
C. An error condition occurs.

I$ReadLn - Read a text line with editing

ASSEMBLER CALL:	OS9 I$ReadLn
MACHINE CODE:	103F 8B
INPUT:	(X) = Address to store data. (Y) = Number of bytes to read. (A) = Path number.
OUTPUT:	(Y) = Actual number of bytes read.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call is the same as I$Read except it reads data from the input file or device until a carriage return character is encountered or until the maximum byte count specified is reached, and that line editing will occur on SCFMAN-type devices. Line editing refers to backspace, line delete, echo automatic line feed, etc.

SCFMAN requires that the last byte entered be an end-of-record character (normally carriage return). If more data is entered than the maximum specified, it will not be accepted and a PD.OVF character (normally bell) will be echoed.

After all data in the file has been read, the next I$ReadLn service request will return an end of file error.

NOTE: For more information on line editing, see 7.1.

I$Seek - Reposition the logical file pointer

ASSEMBLER CALL:	OS9 I$Seek
MACHINE CODE:	103F 88
INPUT:	(A) = Path number. (X) = M.S. 16 bits of desired file position. (U) = L.S. 16 bits of desired file position.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system call repositions the path's "file pointer"; which is the 32-bit addres of the next byte in the file to be read from or written to.

A seek may be performed to any value even if the file is not large enough. Subsequent WRITEs will automatically expand the file to the required size (if possible),

but READs will return an end-of-file condition. Note that a SEEK to address zero is the same as a "rewind" operation.

Seeks to non-random access devices are usually ignored and return without error.

I$SetStt - Set file/device status

ASSEMBLER CALL:	OS9 I$SetStt
MACHINE CODE:	103F 8E
INPUT:	(A) = Path number. (B) Function code. (Other registers depend upon function code)
OUTPUT:	(depends upon function code)
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This system is a "wild card" call used to handle individual device parameters that:

a. are not uniform on all devices

b. are highly hardware dependent

c. need to be user-changeable

The exact operation of this call depends on the device driver and file manager associated with the path. A typical use is to set a terminal's parameters for backspace character, delete character, echo on/off, null padding, paging, etc. It is commonly used in conjunction with the I$GetStt service request which is used to read the device operating parameters. Below are presently defined function codes:

MNEMONIC	CODE	FUNCTION
SS.Opt	$0	Write the 32 byte option section of the path descriptor
SS.Size	$2	Set the file size (RBF)
SS.Reset	$3	Restore head to track zero (RBF)
SS.WRT	$4	Write (format) track (RBF)
SS.Feed	$9	Issue Form Feed (SCF)
SS.FRZ	$A	Freeze DD. Information (RBF)
SS.SPT	$B	Set Sectors per track (RBF)
SS.SQD	$C	Sequence down disk drive (RBF)
SS.Dcmd	$D	Direct command to hard disk controller (RBF)

Codes 128 through 255 their parameter passing conventions are user definable (see the sections of this manual on writing device drivers). The function code and register stack are passed to the device driver.

SS.Opt (code 0):	Write option section of the path descriptor.
INPUT:	(A) = Path number (B) = Function code 0 (X) = Address of a 32 byte status packet.
OUTPUT:	None.

This setstat function writes the option section of the path descriptor from the 32 byte status packet pointed to by the X register. It is typically used to set the device operating parameters, such as echo, auto line feed, etc.

SS.Size (code 2):	Set the file size (RBFMAN-type devices)
INPUT:	(A) = Path number (B) = Function code 2 (X) = M.S. 16 bits of desired file size. (U) = L.S. 16 bits of desired file size.
OUTPUT:	None.

This setstat function is used to change the file's size.

SS.Reset (code 3):	Restore head to track zero.
INPUT:	(A) = Path number (B) = Function code 3
OUTPUT:	None.

Home disk head to track zero. Used for formatting and for error recovery.

SS.WTrk (code 4):	Write track
INPUT:	(A) = Path number (B) = Function code 4 (X) = Address of track buffer. (U) = Track number (L.S. 8 bits) (Y) = Side/density Bit B0 = SIDE (0 = side zero, 1 = side one) Bit B1 = DENSITY (0 = single, 1 = double)
OUTPUT:	None.

This code causes a format track (most floppy disks) operation to occur. For hard disks or floppy disks with a "format entire disk" command, this command should format the entire media only when the track number equals zero.

SS.FRZ (code $A):	Freeze DD. Information
INPUT:	none
OUTPUT:	none

Inhibits the reading of identification sector (LSN 0) to DD.xxx variables (that define disk formats) so non-standard disks may be read.

SS.SPT (code $B):	Set Sectors Per Track
INPUT:	X = new sectors per track

I$Write - Write data to file or device

ASSEMBLER CALL:	OS9 I$Write
MACHINE CODE:	103F 8A
INPUT:	(X) = Address of data to write. (Y) = Number of bytes to write. (A) = Path number.
OUTPUT:	(Y) = Number of bytes actually written.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

I$Write outputs one or more bytes to a file or device associated with the path number

specified. The path must have been OPENed or CREATEd in the WRITE or UPDATE access modes.

Data is written to the file or device without processing or editing. If data is written past the present end-of-file, the file is automatically expanded.

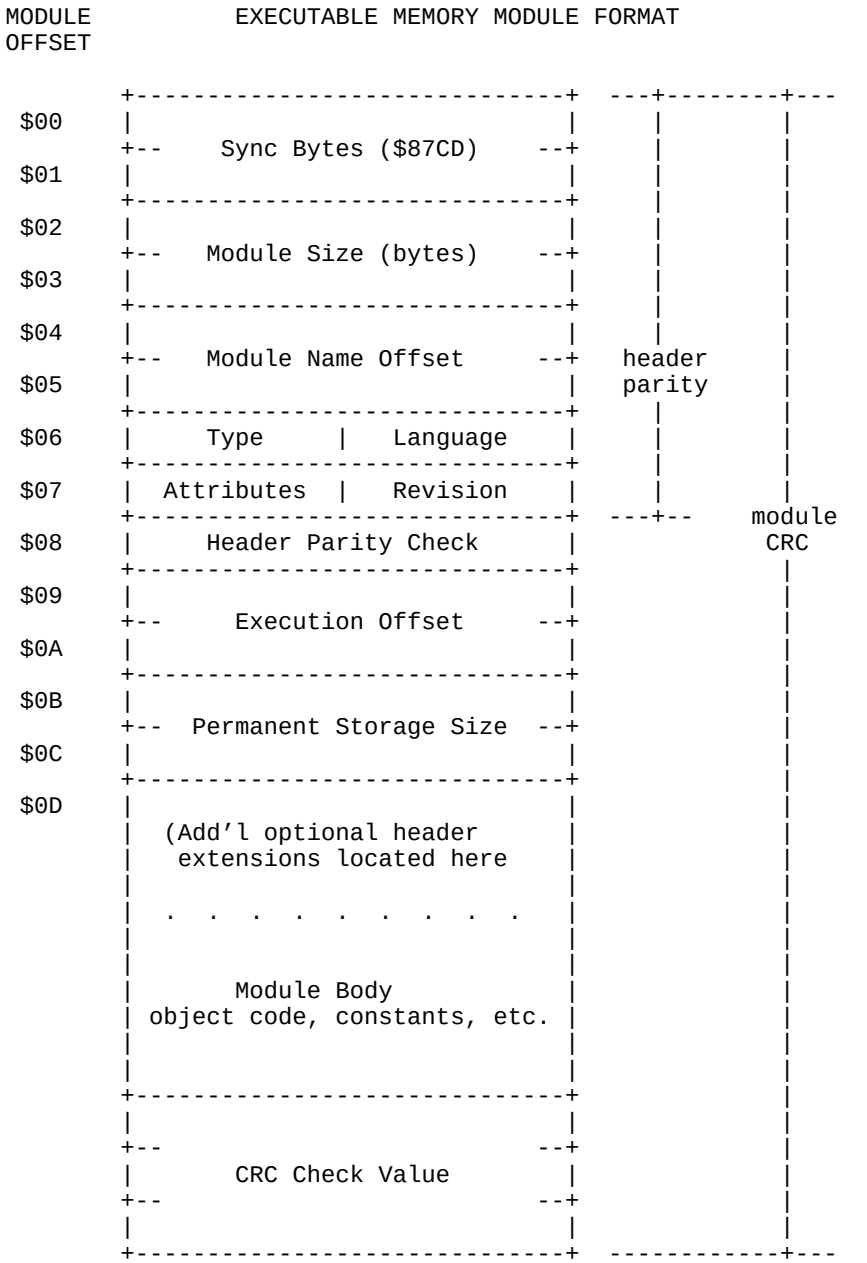
I$WritLn - Write line of text with editing

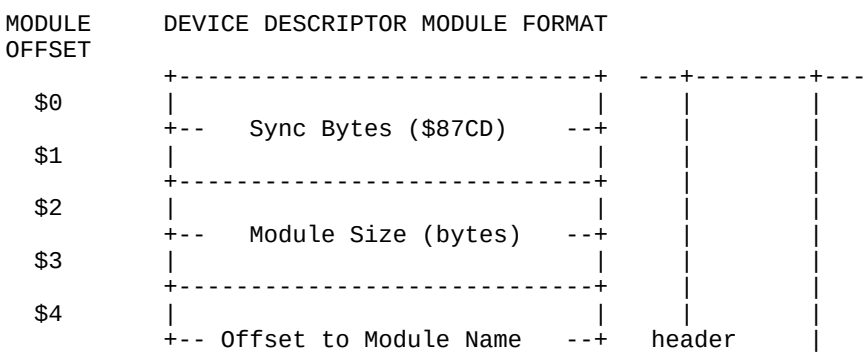
ASSEMBLER CALL:	OS9 I$WritLn
MACHINE CODE:	103F 8C
INPUT:	(X) = Address of data to write. (Y) = Maximum number of bytes to write. (A) = Path number.
OUTPUT:	(Y) = Actual number of bytes written.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

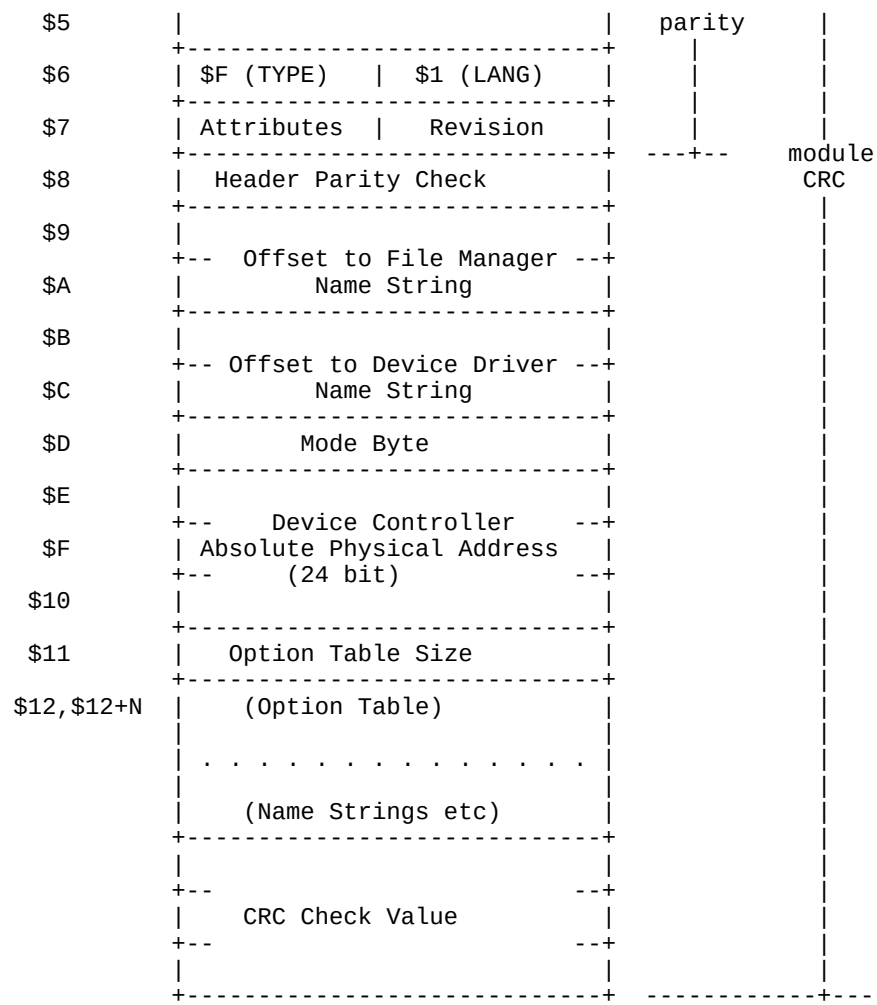
This system call is similar to I$Write except it writes data until a carriage return character is encountered. Line editing is also activated for character-oriented devices such as terminals, printers, etc. The line editing refers to auto line feed, null padding at end of line, etc.

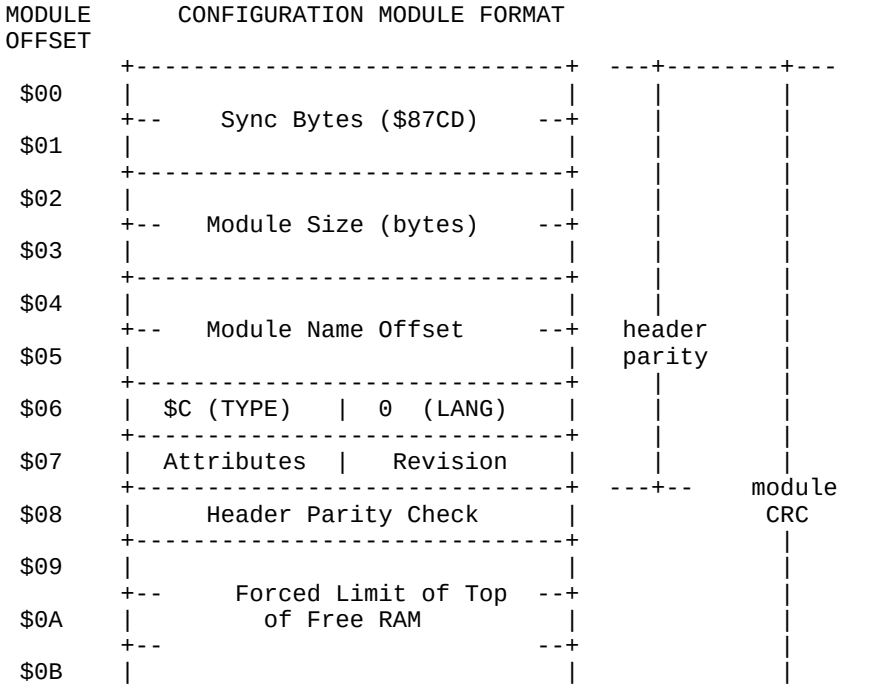
For more information about line editing, see section 7.1.

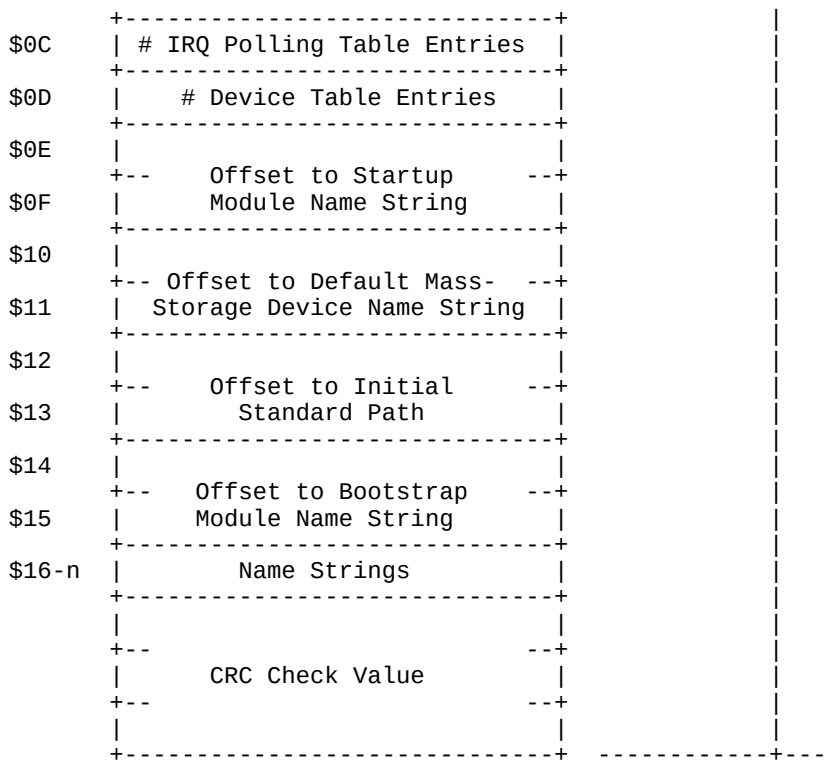
Appendix A. Memory Module Diagrams






$5
$6
$7
$8
$9
$A
$B
$C
$D
$E
$F
$10
$11
$12,$12+N
parity
module
CRC
$F (TYPE)
$1 (LANG)
Attributes
Revision
Header Parity Check
Offset to File Manager
Name String
Offset to Device Driver
Name String
Mode Byte
Device Controller
Absolute Physical Address
(24 bit)
Option Table Size
(Option Table)
(Name Strings etc)
CRC Check Value


MODULE OFFSET
CONFIGURATION MODULE FORMAT
$00
$01
$02
$03
$04
$05
$06
$07
$08
$09
$0A
$0B
Sync Bytes ($87CD)
Module Size (bytes)
Module Name Offset
header
parity
$C (TYPE)
0 (LANG)
Attributes
Revision
module
CRC
Header Parity Check
Forced Limit of Top
of Free RAM


$0C
IRQ Polling Table Entries
$0D
Device Table Entries
$0E
Offset to Startup
$0F
Module Name String
$10
Offset to Default Mass-
$11
Storage Device Name String
$12
Offset to Initial
$13
Standard Path
$14
Offset to Bootstrap
$15
Module Name String
$16-n
Name Strings
CRC Check Value

Appendix B. Standard Floppy Disk Formats

Table B-1. Single Density Floppy Disk Format

SIZE	5"		8"	
DENSITY	SINGLE		SINGLE	
#TRACKS	35		77	
#SECTORS/TRACK	10		16	
BYTES/TRACK (UNFORMATTED)	3125		5208	
FORMAT FIELD	**#BYTES (DEC)**	**VALUE (HEX)**	**#BYTES (DEC)**	**VALUE (HEX)**
HEADER (ONCE PER TRACK)	30	FF	30	FF
	6	00	6	00
	1	FC	1	FC
	12	FF	12	FF
SECTOR (REPEATED N TIMES)	6	00	6	00
	1	FE	1	FE
	1	(TRK #)	1	(TRK #)
	1	(SIDE #)	1	(SIDE #)
	1	(SECT #)	1	(SECT #)
	1	(BYTCNT)	1	(BYTCNT)
	2	(CRC)	2	(CRC)
	10	FF	10	FF
	6	00	6	00
	1	FB	1	FB
	256	(DATA)	256	(DATA)
	2	(CRC)	2	(CRC)
	10	FF	10	FF
TRAILER (ONCE PER TRACK)	96	FF	391	FF
BYTES/SECTOR (FORMATTED)	256		256	
BYTES/TRACK (FORMATTED)	2560		4096	
BYTES/DISK (FORMATTED)	89,600		315,392	

Table B-2. Double Density Floppy Disk Format

SIZE	5"	8"
DENSITY	DOUBLE	DOUBLE
#TRACKS	35	77
#SECTORS/TRACK	16	28

BYTES/TRACK (UNFORMATTED)	6250		10,416	
FORMAT FIELD	**#BYTES (DEC)**	**VALUE (HEX)**	**#BYTES (DEC)**	**VALUE (HEX)**
HEADER (ONCE PER TRACK)	80	4E	80	4E
	12	00	12	00
	3	F5 (A1)	3	F5
	1	FC	1	FC
	32	4E	32	4E
SECTOR (REPEATED N TIMES)	12	00	12	00
	3	F5 (A1)	3	F5
	1	FE	1	FE
	1	(TRK #)	1	(TRK #)
	1	(SIDE #)	1	(SIDE #)
	1	(SECT #)	1	(SECT #)
	1	(BYTCNT)	1	(BYTCNT)
	2	(CRC)	2	(CRC)
	22	4E	22	4E
	12	00	12	00
	3	F5 (A1)	3	F5 (A1)
	1	FB	1	FB
	256	(DATA)	256	(DATA)
	2	(CRC)	2	(CRC)
	22	4E	22	4E
TRAILER (ONCE PER TRACK)	682	4E	768	4E
BYTES/SECTOR (FORMATTED)	256		256	
BYTES/TRACK (FORMATTED)	4096		7168	
BYTES/DISK (FORMATTED)	141,824		548,864	

Appendix C. Service Request Summary

Table C-1. User Mode Service Requests

Code	Mnemonic	Function	Page
103F 00	F$LINK	Link to memory module.	
103F 01	F$LOAD	Load module(s) from a file.	
103F 02	F$Unlink	Unlink a module.	
103F 03	F$Fork	Create a new process.	
103F 04	F$Wait	Wait for child process to die.	
103F 05	F$Chain	Load and execute a new primary module	
103F 06	F$Exit	Terminate the calling process.	
103F 07	F$Mem	Resize data memory area,	
103F 08	F$Send	Send a signal to another process,	
103F 09	F$ICPT	Set up a signal intercept trap.	
103F 0A	F$Sleep	Put calling process to sleep.	
103F 0C	F$ID	Get process ID / user ID	
103F 0D	F$SPrior	Set process priority.	
103F 0E	F$SSWI	Set SWI vector.	
103F 0F	F$PErr	Print error message.	
103F 10	F$PrsNam	Parse a path name,	
103F 11	F$CmpNam	Compare two names	
103F 12	F$SchBit	Search bit map for a free area	
103F 13	F$AllBit	Set bits in an allocation bit map	
103F 14	F$DelBit	Deallocate in a bit map	
103F 15	F$Time	Get system date and time.	
103F 16	F$STime	Set system date and time.	
103F 17	F$CRC	Compute CRC	
103F 18	F$GPrDsc	Get Process Descriptor copy	
103F 19	F$GBlkMp	Get system Block Map copy	
103F 1A	F$GModDr	Get Module Directory copy	
103F 1B	F$CpyMem	Copy external Memory	
103F 1C	F$SUser	Set User ID number	
103F 1D	F$UnLoad	Unlink module by name	

Table C-2. System Mode Privileged Service Requests

Code	Mnemonic	Function	Page
103F 28	F$SRqMem	System memory request	
103F 29	F$SRtMem	System memory return	
103F 2A	F$IRQ	Add or remove device from IRQ table.	
103F 2B	F$IOQU	Enter I/O queue	
103F 2C	F$AProc	Insert process in active process queue	

Code	Mnemonic	Function	Page
103F 2D	F$NProc	Start next process	
103F 2E	F$VModul	Validate module	
103F 2F	F$Find64	Find a 64 byte memory block	
103F 30	F$All64	Allocate a 64 byte memory block	
103F 31	F$Ret64	Deallocate a 64 byte memory block	
103F 32	F$SSVC	Install function request	
103F 33	F$IODel	Delete I/O device from system	
103F 34	F$SLink	System Link	
103F 35	F$Boot	Bootstrap system	
103F 36	F$BtMem	Bootstrap Memory request	
103F 37	F$GProcP	Get Process Pointer	
103F 38	F$Move	Move data (low bound first)	
103F 39	F$AllRAM	Allocate RAM blocks	
103F 3A	F$AllImg	Allocate Image RAM blocks	
103F 3B	F$DelImg	Deallocate Image RAM blocks	
103F 3C	F$SetImg	Set process DAT Image	
103F 3D	F$FreeLB	get Free Low block	
103F 3E	F$FreeHB	get Free High block	
103F 3F	F$AllTsk	Allocate process Task number	
103F 40	F$DelTsk	Deallocate process Task number	
103F 41	F$SetTsk	Set process Task DAT registers	
103F 42	F$ResTsk	Reserve Task number	
103F 43	F$RelTsk	Release Task number	
103F 44	F$DATLog	Convert DAT block/offset to Logical Addr	
103F 45	F$DATTmp	Make Temporary DAT image	
103F 46	F$LDAXY	Load A [X, [Y]]	
103F 47	F$LDAXYP	Load A [X+, [Y]]	
103F 48	F$LDDDXY	Load D [D+X, [Y]]	
103F 49	F$LDABX	Load A from 0,1 in task B	
103F 4A	F$STABX	Store A at 0,X in task B	
103F 4B	F$AllPrc	Allocate Process descriptor	
103F 4C	F$DelPrc	Deallocate Process descriptor	
103F 4D	F$ELink	Link using module directory Entry	
103F 4E	F$FModul	Find Module directory entry	
103F 4F	F$MapBlk	Map specific Block	
103F 50	F$ClrBlk	Clear specific Block	
103F 51	F$DelRam	Deallocate RAM blocks	

Table C-3. Input/Output Service Requests

Code	Mnemonic	Function	Page
103F 80	I$Attach	Attach a new device to the system.	

Code	Mnemonic	Function	Page
103F 81	I$Detach	Remove a device from the system.	
103F 82	I$Dup	Duplicate a path.	
103F 83	I$Create	Create a path to a new file.	
103F 84	I$Open	Open a path to a file or device	
103F 85	I$MakDir	Make a new directory	
103F 86	I$ChgDir	Change working directory.	
103F 87	I$Delete	Delete a file.	
103F 88	I$Seek	Reposition the logical file pointer	
103F 89	I$Read	Read data from a file or device	
103F 8A	I$Write	Write Data to File or Device	
103F 8B	I$ReadLn	Read a text line with editing.	
103F 8C	I$WritLn	Write Line of Text with Editing	
103F 8D	I$GetStt	Get file device status.	
103F 8E	I$SetStt	Set file/device status	
103F 8F	I$Close	Close a path to a file/device.	
103F 90	I$DeletX	Delete a file	

Table C-4. Standard I/O Paths

0 = Standard Input
1 = Standard Output
2 = Standard Error Output

Table C-5. Module Types

$1	Program
$2	Subroutine module
$3	Multi-module
$4	Data module
$C	System Module
$D	File Manager
$E	Device Driver
$F	Device Descriptor

Table C-6. File Access Codes

READ	$01
WRITE	$02
UPDATE	READ + WRITE
EXEC	$04
PREAD	$08
PWRIT	$10
PEXEC	$20

SHARE	$40
DIR	$80

Table C-7. Module Languages

$0	Data
$1	6809 Object code
$2	BASIC09 I-code
$3	Pascal P-Code
$4	C I-code
$5	Cobol I-code
$6	Fortan I-code
$7	6309 Object code

Table C-8. Module Attributes

$8	Reentrant

Appendix D. Error Codes

OS-9 Error Codes

The error codes are shown both in hexadecimal (first column) and decimal (second column). Error codes other than those listed are generated by programming languages or user programs.

HEX	DEC	
$C8	200	PATH TABLE FULL - The file cannot be opened because the system path table is currently full.
$C9	201	ILLEGAL PATH NUMBER - Number too large or for non-existant path.
$CA	202	INTERRUPT POLLING TABLE FULL
$CB	203	ILLEGAL MODE - attempt to perform I/O function of which the device or file is incapable.
$CC	204	DEVICE TABLE FULL - Can't add another device
$CD	205	ILLEGAL MODULE HEADER - module not loaded because its sync code, header parity, or CRC is incorrect.
$CE	206	MODULE DIRECTORY FULL - Can't add another module
$CF	207	MEMORY FULL - Level One: not enough contiquous RAM free. Level Two: process address space full
$D0	208	ILLEGAL SERVICE REQUEST - System call had an illegal code number
$D1	209	MODULE BUSY - non-sharable module is in use by another process.
$D2	210	BOUNDARY ERROR - Memory allocation or deallocation request not on a page boundary.
$D3	211	END OF FILE - End of file encountered on read.
$D4	212	RETURNING NON-ALLOCATED MEMORY - (NOT YOUR MEMORY) attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT - device has damaged file structure.
$D6	214	NO PERMISSION - file attributes do not permit access requested.
$D7	215	BAD PATH NAME - syntax error in pathlist (illegal character, etc.).
$D8	216	PATH NAME NOT FOUND - can't find pathlist specified.
$D9	217	SEGMENT LIST FULL - file is too fragmented to be expanded further.
$DA	218	FILE ALREADY EXISTS - file name already appears in current directory.
$DB	219	ILLEGAL BLOCK ADDRESS - device's file structure has been damaged.
$DC	220	ILLEGAL BLOCK SIZE - device's file structure has been damaged.
$DD	221	MODULE NOT FOUND - request for link to module not found in directory.

HEX	DEC	
$DE	222	SECTOR OUT OF RANGE - device file structure damaged or incorrectly formatted.
$DF	223	SUICIDE ATTEMPT - request to return memory where your stack is located.
$E0	224	ILLEGAL PROCESS NUMBER - no such process exists.
$E2	226	NO CHILDREN - can't wait because process has no children.
$E3	227	ILLEGAL SWI CODE - must be 1 to 3.
$E4	228	PROCESS ABORTED - process aborted by signal code 2.
$E5	229	PROCESS TABLE FULL - can't fork now.
$E6	230	ILLEGAL PARAMETER AREA - high and low bounds passed in fork call are incorrect.
$E7	231	KNOWN MODULE - for internal use only.
$E8	232	INCORRECT MODULE CRC - module has bad CRC value.
$E9	233	SIGNAL ERROR - receiving process has previous unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE - unable to locate module.
$EB	235	BAD NAME - illegal name syntax.
$EC	236	BAD HEADER - module header parity incorrect
$ED	237	RAM FULL - no free system RAM available at this time
$EE	238	UNKNOWN PROCESS ID - incorrect process ID number
$EF	239	NO TASK NUMBER AVAILABLE - all task numbers in use

Device Driver/Hardware Errors

The following error codes are generated by I/O device drivers, and are somewhat hardware dependent. Consult manufacturer's hardware manual for more details.

HEX	DEC	
$F0	240	UNIT ERROR - device unit does not exist
$F1	241	SECTOR ERROR - sector number is out of range.
$F2	242	WRITE PROTECT - device is write protected.
$F3	243	CRC ERROR - CRC error on read or write verify
$F4	244	READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
$F5	245	WRITE ERROR - hardware error during disk write operation.
$F6	246	NOT READY - device has "not ready" status.
$F7	247	SEEK ERROR - physical seek to non-existant sector.
$F8	248	MEDIA FULL - insufficient free space on media.
$F9	249	WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
$FA	250	DEVICE BUSY - non-sharable device is in use
$FB	251	DISK ID CHANGE - Media was changed with files open

HEX	DEC	
$FC	252	RECORD IS LOCKED-OUT - Another process is accessing the requested record.
$FD	253	NON-SHARABLE FILE BUSY - Another process is accessing the requested file.
$FE	254	I/O DEADLOCK ERROR - Two processes are attempting to use the same two disk areas simultaneously.

Appendix E. Level Two System Service Requests

F$AllImg - Allocate Image RAM blocks

ASSEMBLER CALL:	OS9 F$AllImg
MACHINE CODE:	103F 3A
INPUT:	(A) = Beginning block number (B) = Number of blocks (X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Allocates RAM blocks for process DAT image. The blocks do not need to be contiguous.

Note: This is a privileged system mode service request.

F$AllPrc - Allocate Process descriptor

ASSEMBLER CALL:	OS9 F$AllPrc
MACHINE CODE:	103F 4B
INPUT:	none
OUTPUT:	(U) = Process Descriptor pointer
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Allocates and initializes a 512-byte process descriptor.

Note: This is a privileged system mode service request.

F$AllRAM - Allocate RAM blocks

ASSEMBLER CALL:	OS9 F$AllRAM
MACHINE CODE:	103F 39
INPUT:	(B) = Desired block count
OUTPUT:	(D) = Beginning RAM block number
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Searches the Memory Block map for the desired number of contuguous free RAM blocks.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$AllTsk - Allocate process Task number

ASSEMBLER CALL:	OS9 F$AllTsk
MACHINE CODE:	103F 3F
INPUT:	(X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Allocates a Task number for the given process.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$Boot - Bootstrap system

ASSEMBLER CALL:	OS9 F$Boot
MACHINE CODE:	103F 35
INPUT:	none
OUTPUT:	none
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Links to the module named “Boot” or as specified in the INIT module; calls linked module; and expects the return of a pointer and size of an area which is then searched for new modules.

Note: This is a privileged system mode service request.

F$BtMem - Bootstrap Memory request

ASSEMBLER CALL:	OS9 F$BtMem
MACHINE CODE:	103F 36
INPUT:	(D) = Byte count requested.
OUTPUT:	(D) = Byte count granted. (U) = Pointer to memory allocated.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Allocates requested memory (rounded up to nearest block) as contigous memory in the system’s address space.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$ClrBlk - Clear specific Block

ASSEMBLER CALL:	OS9 F$ClrBlk
MACHINE CODE:	103F 50
INPUT:	(B) = Number of blocks (U) = Address of first block

OUTPUT:	None.
ERROR OUTPUT:	None.

Marks blocks in process DAT image as unallocated.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$CpyMem - Copy external Memory

ASSEMBLER CALL:	OS9 F$CpyMem
MACHINE CODE:	103F 1B
INPUT:	(D) = Starting Memory Block number (X) = Offset in block to begin copy (Y) = Byte count (U) = Caller's destination buffer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Reads external memory into the user's buffer for inspection. Any memory in the system may be viewed in this way.

F$DATLog - Convert DAT block/offset to Logical Addr

ASSEMBLER CALL:	OS9 F$DATLog
MACHINE CODE:	103F 44
INPUT:	(B) = DAT image offset (X) = Block offset
OUTPUT:	(X) = Logical address.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Converts a DAT imaqe block number and block offset to its equivalent logical address.

Note: This is a privileged system mode service request.

F$DATTmp - Make Temporary DAT image

ASSEMBLER CALL:	OS9 F$DATTmp
MACHINE CODE:	103F 45
INPUT:	(D) = Block number
OUTPUT:	(Y) = DAT image pointer
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Builds a temporary DAT image to access the given memory block.

NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$DelImg - Deallocate Image RAM blocks

ASSEMBLER CALL:	OS9 F$DelImg
MACHINE CODE:	103F 3B
INPUT:	(A) = Beginning block number (B) = Block count (X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Deallocated memory from the process' address space.

Note: This is a privileged system mode service request.

F$DelPrc - Deallocate Process descriptor

ASSEMBLER CALL:	OS9 F$DelPrc
MACHINE CODE:	103F 4C
INPUT:	(A) = Process ID.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Returns process descriptor memory to system free memory pool.

Note: This is a privileged system mode service request.

F$DelRam - Deallocate RAM blocks

ASSEMBLER CALL:	OS9 F$DelRam
MACHINE CODE:	103F 51
INPUT:	(B) = Number of blocks (X) = Beginning block number.
OUTPUT:	None.
ERROR OUTPUT:	None.

Marks blocks in system memory block map as unallocated.

Note: This is a privileged system mode service request.

F$DelTsk - Deallocate process Task number

ASSEMBLER CALL:	OS9 F$DelTsk

MACHINE CODE:	103F 40
INPUT:	(X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Releases the Task number in use by the process.

Note: This is a privileged system mode service request.

F$ELink - Link using module directory Entry

ASSEMBLER CALL:	OS9 F$ELink
MACHINE CODE:	103F 4D
INPUT:	(B) = Module type. (X) = Pointer to module directory entry.
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Performs a "Link" given a pointer to a module directory entry. Note that this call differs from F$Link in that a pointer to the module directory entry is supplied rather than a pointer to a module name.

Note: This is a privileged system mode service request.

F$FModul - Find Module directory entry

ASSEMBLER CALL:	OS9 F$FModul
MACHINE CODE:	103F 4E
INPUT:	(A) = Module type. (X) = Module Name string pointer. (Y) = Name string DAT image pointer.
OUTPUT:	(A) = Module Type. (B) = Module Revision. (X) = Updated past name string. (U) = Module directory entry pointer.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This call returns a pointer to the module directory entry given the module name.

Note: This is a privileged system mode service request.

F$FreeHB - Get Free High block

ASSEMBLER CALL:	OS9 F$FreeHB
MACHINE CODE:	103F 3E
INPUT:	(B) = Block count (Y) = DAT image pointer
OUTPUT:	(A) High block number
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Searches the DAT image for the highest free block of given size.

Note: This is a privileged system mode service request.

F$FreeLB - Get Free Low block

ASSEMBLER CALL:	OS9 F$FreeLB
MACHINE CODE:	103F 3D
INPUT:	(B) = Block count (Y) = DAT image pointer
OUTPUT:	(A) = Low block number
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Searches the DAT image for the lowest free block of given size.

Note: This is a privileged system mode service request.

F$GBlkMp - Get system Block Map copy

ASSEMBLER CALL:	OS9 F$GBlkMp
MACHINE CODE:	103F 19
INPUT:	(X) = 1024 byte buffer pointer
OUTPUT:	(D) = Number of bytes per block (MMU block size dependent). (Y) = Size of system's memory block map.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Copies the system's memory block map into the user's buffer for inspection.

F$GModDr - Get Module Directory copy

ASSEMBLER CALL:	OS9 F$GModDr
MACHINE CODE:	103F 1A
INPUT:	(X) = 2048 byte buffer pointer

ERROR OUTPUT: (CC) = C bit set. (B) = Appropriate error code.

Copies the system's module directory into the user's buffer for inspection.

F$GPrDsc - Get Process Descriptor copy

ASSEMBLER CALL: OS9 F$GPrDsc
MACHINE CODE: 103F 18
INPUT: (A) = Requested process ID. (X) = 512 byte buffer pointer.
OUTPUT: None.
ERROR OUTPUT: (CC) = C bit set. (B) = Appropriate error code.

Copies a process descriptor into the calling process' buffer for inspection There is no way to change data in a process descriptor.

F$GProcP - Get Process Pointer

ASSEMBLER CALL: OS9 F$GProcP
MACHINE CODE: 103F 37
INPUT: (A) = Process ID
OUTPUT: (Y) = Pointer to Process Descriptor
ERROR OUTPUT: (CC) = C bit set. (B) = Appropriate error code.

Translates a process ID number to the address of its process descriptor in the system address space.

Note: This is a privileged system mode service request.

F$LDABX - Load A from 0,1 in task B

ASSEMBLER CALL: OS9 F$LDABX
MACHINE CODE: 103F 49
INPUT: (B) = Task number (X) = Data pointer
OUTPUT: (A) = Data byte at 0,X in task's address space
ERROR OUTPUT: (CC) = C bit set. (B) = Appropriate error code.

One byte is returned from the logical address in (X) in the given task's address space. This is typically used to get one byte from the current process's memory in a system state routine.

Note: This is a privileged system mode service request.

F$LDAXY - Load A [X, [Y]]

ASSEMBLER CALL:	OS9 F$LDAXY
MACHINE CODE:	103F 46
INPUT:	(X) = Block offset (Y) = DAT image pointer
OUTPUT:	(A) = data byte at (X) offset of (Y)
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Returns one data byte in the memory block specified by the DAT image in (Y), offset by (X).

Note: This is a privileged system mode service request.

F$LDAXYP - Load A [X+, [Y]]

ASSEMBLER CALL:	OS9 F$LDAXYP
MACHINE CODE:	103F 47
INPUT:	(X) = Block offset (Y) = DAT image pointer
OUTPUT:	(A) = Data byte at (X) offset of (Y) (X) = incremented by one
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Similar to the assembly instruction “LDA ,X+”, except that (X) refers to an offset in the memory block described by the DAT image at (Y).

Note: This is a privileged system mode service request.

F$LDDDXY - Load D [D+X, [Y]]

ASSEMBLER CALL:	OS9 F$LDDDXY
MACHINE CODE:	103F 48
INPUT:	(D) = Offset to offset (X) = Offset (Y) = DAT image pointer
OUTPUT:	(D) = bytes address by [D+X,Y]
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Loads two bytes from the memory block described by the DAT image pointed to by (Y). The bytes loaded are at the offset (D+X) in the memory block.

Note: This is a privileged system mode service request.

F$MapBlk - Map specific Block

ASSEMBLER CALL:	OS9 F$MapBlk
MACHINE CODE:	103F 4F
INPUT:	(B) = Number of blocks. (X) = Beginning block number.
OUTPUT:	(U) = Address of first block.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Maps specified block(s) into unallocated blocks of process address space.

Note: This is a privileged system mode service request.

F$Move - Move data (low bound first)

ASSEMBLER CALL:	OS9 F$Move
MACHINE CODE:	103F 38
INPUT:	(A) = Source Task number (B) = Destination Task number (X) = Source pointer (Y) = Byte count (U) = Destination pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Moves data bytes from one address space to anotber, usually from System's to User's, or vice-versa.

Note: This is a privileged system mode service request.

F$RelTsk - Release Task number

ASSEMBLER CALL:	OS9 F$RelTsk
MACHINE CODE:	103F 43
INPUT:	(B) = Task number
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Releases the specified DAT Task number.

Note: This is a privileged system mode service request.

F$ResTsk - Reserve Task number

DAT task number.

ASSEMBLER CALL:	OS9 F$ResTsk
MACHINE CODE:	103F 42
INPUT:	none
OUTPUT:	(B) = Task number
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Finds a free DAT task number.

Note: This is a privileged system mode service request.

F$SetImg - Set process DAT Image

ASSEMBLER CALL:	OS9 F$SetImg
MACHINE CODE:	103F 3C
INPUT:	(A) = Beginning image block number (B) = Block count (X) = Process Descriptor pointer (U) New image pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Copies a DAT image into the process descriptor.
NOTE: THIS IS A PRIVILEGED SYSTEM MODE SERVICE REQUEST

F$SetTsk - Set process Task DAT registers

ASSEMBLER CALL:	OS9 F$SetTsk
MACHINE CODE:	103F 41
INPUT:	(X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Sets the process Task DAT registers.

Note: This is a privileged system mode service request.

F$SLink - System Link

ASSEMBLER CALL:	OS9 F$SLink
MACHINE CODE:	103F 34

INPUT:	(A) = Module Type. (X) = Module Name string pointer. (Y) = Name string DAT image pointer.
OUTPUT:	(A) = Module Type. (B) = Module Revision. (X) = Updated Name string pointer. (Y) = Module Entry point. (U) = Module pointer.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Links a module whose name is outside the current (system) process' adress space into the Address space that contains its name.

Note: This is a privileged system mode service request.

F$SRqMem - System Memory Request

ASSEMBLER CALL:	OS9 F$SRqMem
MACHINE CODE:	103F 28
INPUT:	(D) = byte count of requested memory
OUTPUT:	(D) = byte count of memory granted (U) = pointer to memory block allocated
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Allocates the requested memory (rounded up to the nearest page) in the system's address space. Useful for allocating I/O buffers and other semi-permanent system memory.

Note: This is a privileged system mode service request.

F$SRtMem - System Memory Return

ASSEMBLER CALL:	OS9 F$SRtMem
MACHINE CODE:	103F 29
INPUT:	(D) = Byte count of memory being returned (U) = Address of memory block being returned
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Returns system memory (e.g., memory in the system address space) after it is no longer needed.

Note: This is a privileged system mode service request.

F$STABX - Store A at 0,X in task B

ASSEMBLER CALL:	OS9 F$STABX
MACHINE CODE:	103F 4A
INPUT:	(A) = Data byte to store in Task's address space (B) = Task number (X) = Logical address in task's address space to store
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This is similar to the assembly instruction "STA 0,X", except that (X) refers to an address in the given task's address space rather than the current address space.

Note: This is a privileged system mode service request.

F$SUser Set User ID number

ASSEMBLER CALL:	OS9 F$SUser
MACHINE CODE:	103F 1C
INPUT:	(Y) = desired User ID number
OUTPUT:	None
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Alters the current user ID to that specified, without error checking.

F$UnLoad - Unlink module by name

ASSEMBLER CALL:	OS9 F$UnLoad
MACHINE CODE:	103F 1D
INPUT:	(A) = Module Type (X) = Module Name pointer
OUTPUT:	None
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

Locates the module to the module directory, decrements its link count and removes it from the directory if the count reaches zero. Note that this call differs from F$UnLink in that the a pointer to module name is supplied rather than the address of the module header.

I$DeletX - Delete a file

ASSEMBLER CALL:	OS9 I$Deletx
MACHINE CODE:	103F 90
INPUT:	(X) = Address of pathlist (A) = Access mode.

OUTPUT:	(X) = Updated past pathlist (trailing spaces skipped).
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

This service request deletes the file specified by the pathlist. The file must have write permission attributes (public write if not the owner), and reside on a multi-file mass storage device. Attempts to delete devices will result in error.

The access mode is used to specify the current working directory or me current execution directory (but not both) in the absence of a full pathlist. If the access mode is read, write, or update, the current data directory is assumed. If the access mode is execute, the current execution directory is assumed. Note that if a full pathlist (a pathlist beginning with a "/") appears, the access mode is ignored.

ACCESS MODES:

1 = Read
2 = Write
3 = Update (read or write)
4 = Execute