OS-9 Operating System

System Programmer's Manual

OS-9 Operating System: System Programmer's Manual

Revision H

Publication date January 1984

This manual edited by Wes Camden; Contributions by William Phelps.

Chapter 1. Introduction

OS-9 Level One is a versatile multiprogramming/multitasking operating system for computers utilizing the Motorola 6809 microprocessor. OS-9 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.
- Level One version for small and medium sized systems.
- Level Two version for large systems with memory management.

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.

1.1. 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. By no means,however, is OS-9 a direct duplication of UNIX. Even though the system functions and interfaces are generally compatible, OS-9 has been designed for better efficiency, greater reliability, and compact size. OS-9 also pioneers several new concepts 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.

6. OS-9 can be dynamically reconfigured by the user, even while the system is running.

1.2. System Hardware Requirements

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.

OS-9 LEVEL ONE

- 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.

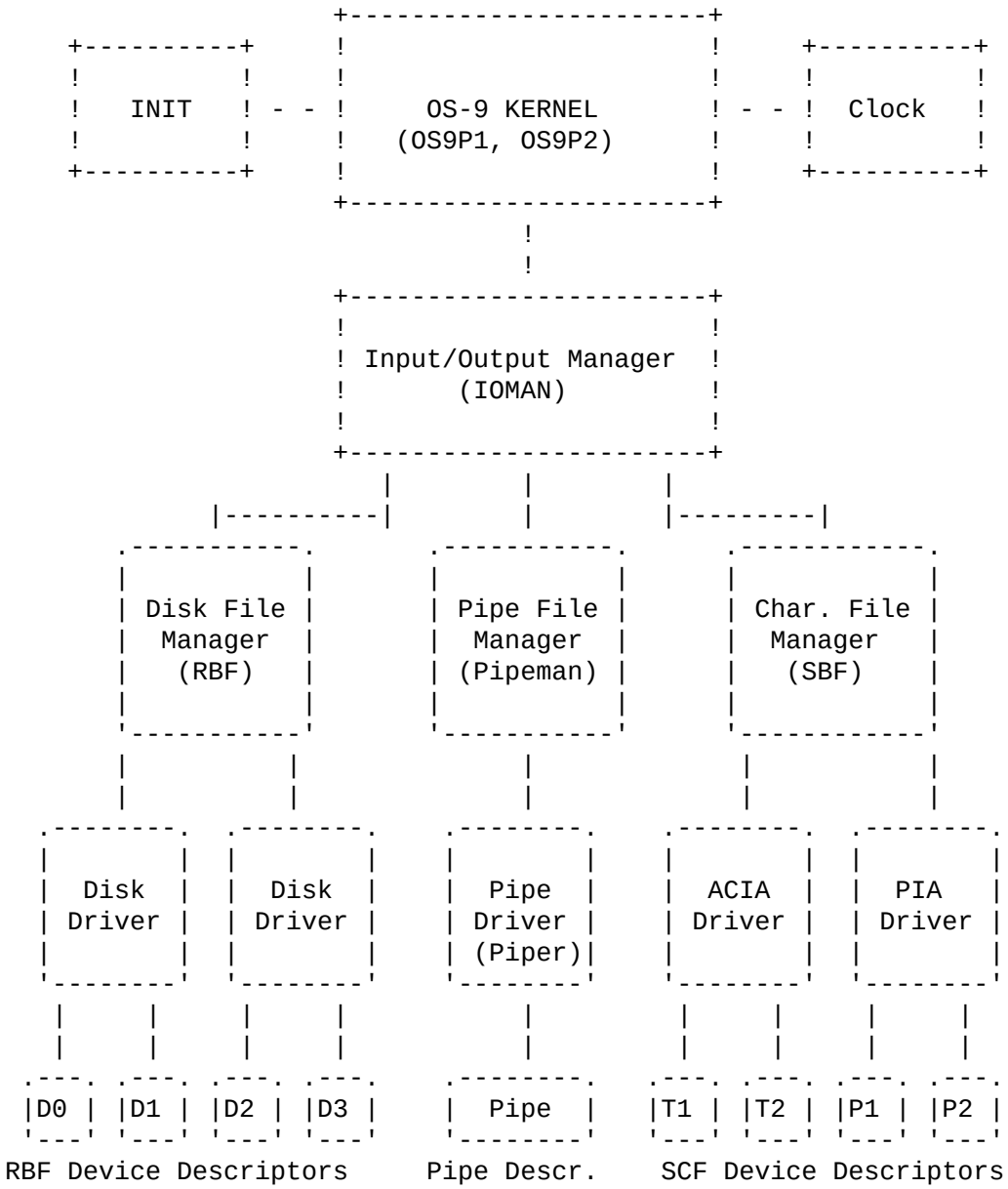
OS-9 LEVEL TWO

- Memory Management Unit with Dynamic Address Translation with selectable memory maps.
- 64K bytes RAM Memory plus approximately 32K per user.
- Console Terminal plus additional user terminal ports.
- Floppy and/or Hard Disk system(s).
- Real Time Clock hardware.
- Optional printer(s) using serial or parallel interfaces.

Chapter 2. Basic System Organization

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

Figure 2.1. OS-9 Component Module Organization



Notice that the diagram on the previous page clearly shows OS-9's multilevel organization.

Level 1 - Kernal and Clock Module

The first level contains the KERNEL, CLOCK MODULE, and INIT. 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.

Level 2 - Input/Output Manager (IOMAN)

The second level is the Input/Output Manager (IOMAN). It provides common processing to all I/O operations. The I/0 Manager is required if any OS-supported I/0 is to be performed.

Level 3 - File Manager Level (RBF, SCF, and Pipeman)

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 (RBF) processes all disk-type device functions, and the Sequential Character File Manager (SCF) 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 SCF or RBF. Another file manager called PIPEMAN supports the “pipe” interprocess communication method that uses memory buffers for data transfer instead of mass storage files.

Level 4 - Device Drivers

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, an ACIA driver for terminals and serial printers, and a PIA driver for parallel printers. Many users add. customized drivers of their own design or purchase drivers from a hardware vendor.

Level 5 - Device Descriptors

The fifth level is the Device Descriptor Level. These modules are small tables that 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 User's 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 called BOOT (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.

The Level One kernel is about 3K bytes long and normally resides in two 2K byte ROMs: “OS9P1” residing at addresses $F800 - $FFFF, and "OS9P2", which is position-independent. OS9P2 only occupies about half (1K) of the ROM, the other space in the ROM is reserved for the disk bootstrap module.

The Level Two kernel is somewhat larger than the Level One version. Its exact size depends on the size of the software routines required for the style of memory management unit used in the particular system. Half of the kernel (called “OS9P1”) resides in ROM with the BOOT module; the other half ("OS9P2") is loaded into RAM with the other OS-9 modules.

The kernel's main functions are:

1. System initialization after reset.
2. Service request processing.
3. Memory management.
4. MPU management (multiprogramming).
5. Interrupt processing.

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

3.1. System Initialization

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. See pages 10.4 and 10.5 for more information on INIT and SYSGO.

OS9p3 is a system module added to version 1.2 of Level 2 that is called during cold start to allow users to define their own system calls. The cold start routine in OS9p2 does a link to OS9p3. If the module exists (in the boot file or in ROM), then OS9p2 will do a BSR to the entry point of the “p3” module. If “p3” does not exist, OS9p2 will centinue with a normal cold start.

Level 2 cannot handle the installation of new OS-9 system calls (via F$SSVC) by user programs because of the separation of system and user address space. F$SSVC requires the service call object code to be in the system address space and expects to receive its address from the service table. OS9p3 is a system module which can be tailored to fit specific needs. The following code is an example of how the OS9p3 module can be used.

```
Microware OS-9 Assembler 2.1   11/18/83   16:06:01       Page 001
OS-9 Level II V1.2, part 3 - OS-9 System Symbol Definitions

00001                             nam OS-9 Level II V1.2, part 3
00002
00003
```

```
00011   ************************************************************
00012   *
00013   *     Module Header
00014   *
00015   00C1            Type     set Systm+Objct
00016   0081            Revs     set ReEnt+1
00017   0000 87CD005E        mod   OS9End,OS9Name,Type,Revs,Cold,256
00018   000D 4F533970   OS9Name  fcs   /OS9p3/
00019

00029   0012 01                  fcb   1          edition number
00030                            use   defsfile
00031   0002            level    equ   2
00032                            opt   -c
00033                            opt   f
00041
00042   ************************************************************
00043   *
00044   *     Routine Cold
00045   *
00046   *
00047
00048   0013 318D0004   Cold     leay  SvcTbl,pcr get service routine
00049   0017 103F32              OS9   F$SSvc     install new service
00050   001A 39                  rts
00051
00052
00053   ************************************************************
00054   *
00055   *     Service Routines Initialization Table
00056   *
00057
00058   0025           F$SayHi  equ   $25        set up new call
00059   *  This should be added to user os9defs file.
00060
00061   001B            SvcTbl   equ   *
00062   001B 25                  fcb   F$SayHi
00063   001C 0001                fdb   SayHi-*-2
00064   001E 80                  fcb $80

Microware OS-9 Assembler 2.1   11/18/83   16:06:04       Page 002
OS-9 Level II V1.2, part 3 - OS-9 System Symbol Definitions
00068          *
00069          *     Service Call Say Hello to user
00070          *
00071          * Input: U = Registers ptr
00072          *            R$X,u = Message ptr (if 0 send default)
00073          *            Max message length = 40 bytes.
00074          *
00075          * Output: Message sent to standard error path of user.
00076          *
00077          * Data: D.Proc
00078          *
00079
00080   001F AE44       SayHi    ldx   R$X,u      get mess. address
00081   0021 2619                bne   SayHi6     bra if not default
00082   0023 109E50              ldy   D.Proc     get proc descr ptr
00083   0026 EE24                ldu   P$SP,y     get caller's stack
```

```
00084   0028 33C8D8             leau  -40,u      room for message
00085   002B 96D0               lda   D.SysTsk   system's task num
00086   002D E626               ldb   P$Task,y   caller's task num
00087   002F 108E0028           ldy   #40        set byte count
00088   0033 308D0012           leax  Hello,pcr  destination ptr
00089   0037 103F38             OS9   F$Move     mess into user mem
00090   003A 30C4               leax  0,u
00091   003C 108E0028   SayHi6  ldy   #40        get max byte count
00092   0040 DE50               ldu   D.Proc     get proc desc ptr
00093   0042 A6C832             lda   P$Path+2,u path num of stderr
00094   0045 103F8C             OS9   I$WritLn   write mess line
00095   0048 39                 rts
00096
00097   0049 48656C6C   Hello   fcc   "Hello there user."
00098   005A 0D                 fcb   $D
00099
00100   005B 5104B6             emod             module CRC
00101
00102   005E            OS9End  equ   *
00103
00104                           end

00000 error(s)
00000 warning(s)
$005E 00094 program bytes generated
$0000 00000 data bytes allocated
$2884 10372 bytes used for symbols
```

3.2. Kernel Service Request Processing

All OS-9 service requests (system calls) are processed via the kernel. Service requests are used to communicate between OS-9 and assembly language 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.

Service requests are divided into two categories:

I/O REQUESTS perform various input/output functions and are passed by the kernel to IOMAN for processing. IOMAN will in turn call the appropriate file manager and device driver modules. 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$”.

A system wide assembly language equate file called “OS9Defs” defines symbolic names for all service requests. The OS9Defs 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 recognized and assembled as the equivalent to:

```
SWI2
```

```
FCB   I$READ
```

3.3. 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” (see chapter 4 for a detailed description of 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. Memory modules are the foundation of OS-9's modular software environment. Some of the 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.

OS-9 Level One uses a software memory management system where all memory is contained within a single 64K memory map. Therefore, OS-9 and all user tasks share a common memory space.

OS-9 Level Two uses memory management hardware that gives OS-9 and each user task a private memory map which can contain up to 64K bytes of memory (depending on the type of MMU hardware). RAM used for data is dynamically assigned to each map on a demand basis. Memory modules are switched into each map when required. One physical memory module can appear simultaneously in several maps if multiple tasks request the same module(s) at the same time.

3.4. Memory Utilization

In Level One, all usable RAM memory must be contiguous from address 0 upward. During the OS-9 start-up sequence, the upper bound of RAM is determined 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. Level Two works similarly except RAM need not be contiguous as the Memory Management Unit (MMU) hardware can dynamically rearrange memory addresses.

All other RAM memory is pooled into a “free memory” space. Memory space is dynamically taken from and returned to the 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. Level Two systems must physically allocate memory according to the size of the smallest memory block the MMU hardware can handle, usually 2048 or 4096 byte blocks. The 256 byte page is still used as a basic allocation size, but OS-9 Level Two must assign a whole number of 2K or 4K blocks. Therefore, Level Two systems use memory most efficiently when program or data memory sizes are close to (but not over) the 2K or 4K block size.

OS-9 automatically assigns memory from the free memory pool whenever any of the following occur:

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

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.

Figure 3.1. Typical Level One Memory Map

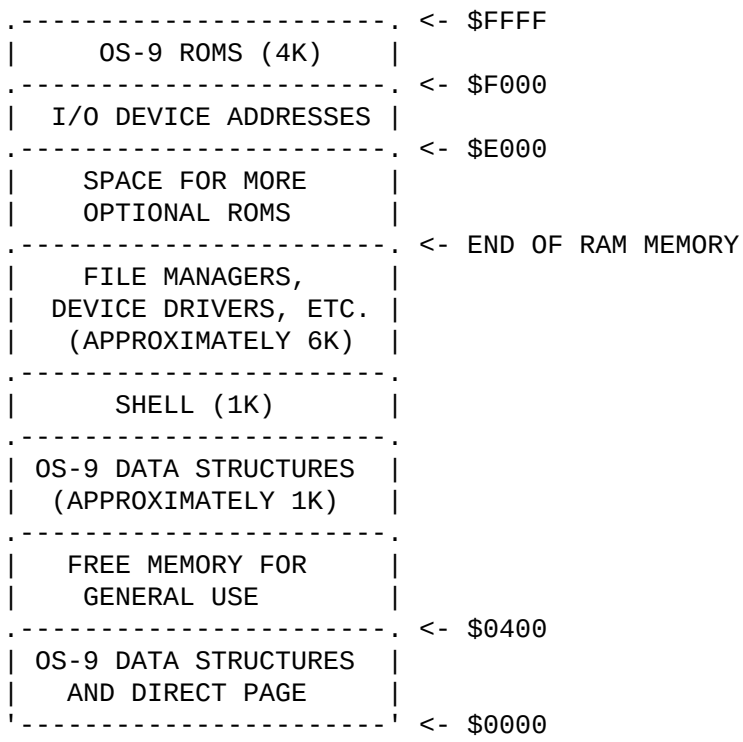
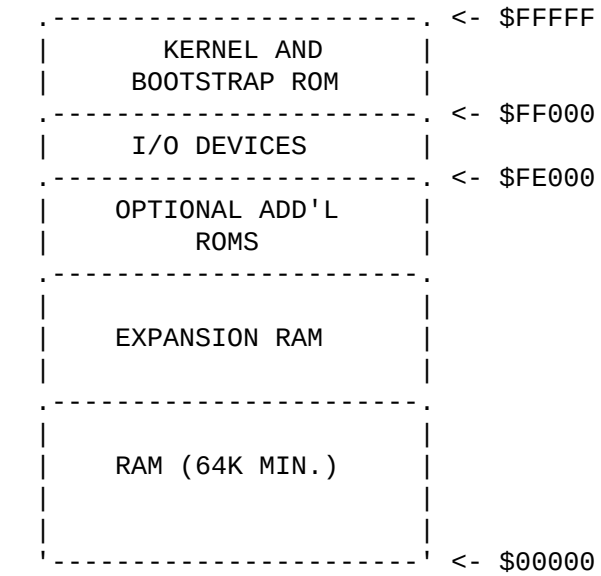


Figure 3.2. Typical Level Two Physical Memory Map



The diagrams above illustrate “typical” systems. Actual memory sizes and addresses may vary depending on the exact system configuration.

3.4.1. Level Two Memory Management Hardware

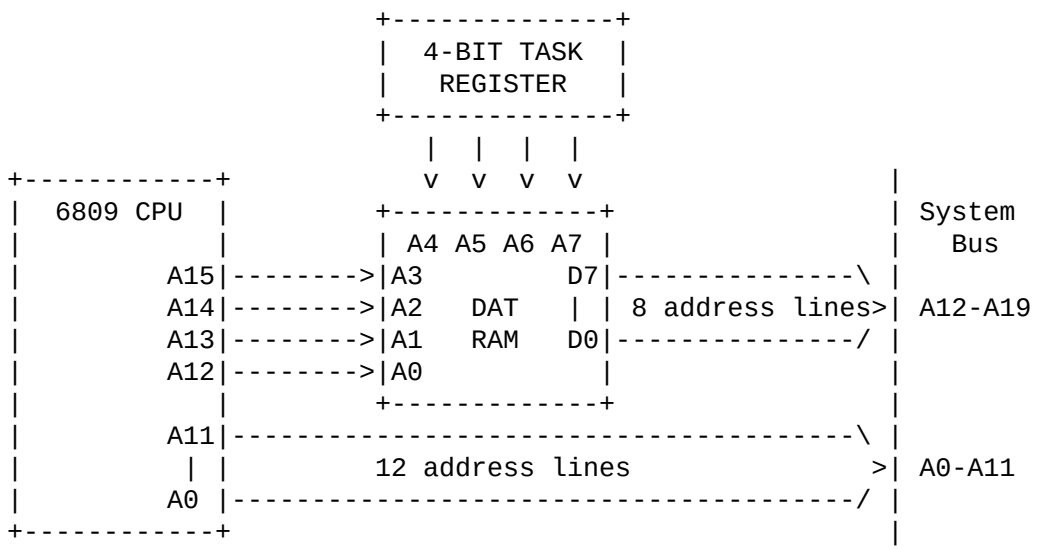
The 6809 CPU produces 16 address lines and in Level One is limited to 64k of address space. OS-9 Level Two uses memory management hardware based on a DAT (Dynamic Address Translation) to allow the CPU to access up to 2 megabytes of memory. The DAT allows the creation of multiple 64k address spaces which can be switched in and out of under the control of OS-9. The maximum amount

of overall system memory and the maximum memory per task depends on the specific design of the memory management unit. Consult the hardware manufacturer's manuals regarding your system's exact specifications.

A physical address space of up to 2 megabytes is addressed by the system bus using 20 or 21 address bits. All RAM, ROM, and I/O devices respond to these extended addresses. The physical address space is further subdivided into “blocks” of either exactly 2K or 4K (depending on the hardware) bytes. The high order 8 or 9 (depending on the hardware) bits of the starting address of the block is called a block number. For example, the block having physical address $3C000 to $3CFFF is called "block 3C".

A logical address space of up to 64K bytes of memory is created for each task (as part of the FORK system call). The address space is assigned blocks of memory as required. Even though memory within the logical address space appears to be contiguous to user programs, it can be constructed from noncontiguous physical blocks because of the address translation function of the DAT. Address spaces can also contain blocks of memory that are common to more than one map.

The diagram below illustrates the design of a *typical* memory management system. For ease of explanation, a system with 16 memory maps with a 4k block block size will be described.



Typical Memory Management Unit Hardware

Not shown in this simplified schematic is the necessary additional circuits which allow the CPU to write data (reading is not necessary) in any word of the mapping RAM.

The DAT is built around a small high speed RAM. In the example above, the size of the DAT RAM is 256 8-bit words. Each word of the mapping RAM is addressed by a combination of the high-order 4 address lines of the CPU and the 4-bit output of the task register. In effect, the task register divides the RAM into 16 sections of 16 words each. Each section defines a 64K logical memory map, and each of the 16 8-bit words it contains corresponds to a 4K memory block. The actual 20-bit physical address sent to the bus is the 8-bit contents of the selected word in the RAM combined with the unaltered low-order 12 bits from the CPU.

Below is an example of how one 16-word section of the DAT is used to define a logical memory map. Assume that the Task Register is set to 3, so we are looking at map 3. (All values below are hexadecimal).

CPU ADDRESS	DAT ADDRESS	DAT DATA	BUS ADDRESS
0xxx	30	15	15xxx
1xxx	31	16	16xxx
2xxx	32	17	17xxx

CPU ADDRESS	DAT ADDRESS	DAT DATA	BUS ADDRESS
3xxx	33	2A	2Axxx
4xxx	34	2D	2Dxxx
5xxx	35	B1	B1xxx
6xxx	36	B2	B2xxx
7xxx	37	FC	FCxxx
8xxx	38	FC	FCxxx
9xxx	39	FC	FCxxx
Axxx	3A	FC	FCxxx
Bxxx	3B	FC	FCxxx
Cxxx	3C	81	81xxx
Dxxx	3D	82	82xxx
Exxx	3E	83	83xxx
Fxxx	3F	FC	FCxxx

Suppose a user program reads a byte from address $40FF. The high order 4 bits are combined with the task register value (currently 3) to produce the DAT RAM address $34. The data contents of this DAT RAM address is $2D. This is combined with the low order 12 bits from the CPU to produce the physical bus address $2D0FF. Likewise, CPU address $D315 would generate physical address $82315.

Notice that several DAT memory locations contain the value $FC. $FC is a special block address that OS-9 commonly uses to represent a place in an address map that currently has no physical memory assigned to it. In fact, some systems have error-detection circuits that detect such illegal memory references by looking for this reserved memory block address. Given that $FC means no memory has been assigned here, by physical memory at logical addresses $0000-$6FFF and $C000-$EFFF. This would be typical of a task with a program module size of about 12K (at $C000-$EFFF) and a data area of 28K (at $0000-$6FFF).

If this task requests OS-9 to allocate an additional 8K of memory to its data area and OS-9 found unassigned free memory blocks at physical addresses $61000-$61FFF and $64000-$64FFF, OS-9's memory allocation function would assign this memory by writing the DAT with values of $61 and $64 in DAT locations $37 and $38, respectively.

Memory map (DAT section) number zero is reserved for OS-9 and the operating system code, data structures, I/O buffers, path descriptors, process descriptors, etc., are all allocated within this map. If this map becomes full, it may not be possible to create new tasks or open new files until other processes terminate or files are closed, even though there may still be considerable free memory left in the system.

3.4.2. DAT Images and Level II System Calls

Typical DAT hardware does not have enough sections to maintain a memory map for each possible process. Therefore, when a process is created a DAT Image is defined in the corresponding process descriptor. The DAT Image is a small array (in this example, 16 bytes) that is an exact copy of what must be written in the DAT RAM to represent the process' memory map. In fact, changes in memory allocation are actually made by changing the DAT Image.

When a process begins its timeslice, a map number (1 .. 15 in this example) corresponding to a memory map, is allocated to it, then the DAT Image in the process descriptor is copied into the DAT map that has been chosen for the process. When a process gives up a time slice the map it was using may be overwritten by another process' if necessary.

User programs should never directly alter the DAT RAM or the DAT Image contents directly or disaster may strike. In fact, in many Level II systems the DAT and DAT images are physically

unaccessable to user programs. Several OS-9 system calls are available to allocate and deallocate memory, or to access data in specific physical memory blocks.

A common case is direct access to I/O devices. Normally, the block(s) that contain I/O devices are kept in OS-9's map and not in user maps, which is proper for true timesharing system security but not for process control applications, for example. In order for a user task to get access to the I/O device block, the “F$MapBlk” system call can be used. It takes a starting block number and block count and maps them into unallocated spaces of the process' address space. You do not have a choice where in your block it is mapped, but the system call returns the logical address where the blocks were inserted. Note that there must be sufficient unused space in the process' map. For example, suppose the I/O block in your system is located at extended addresses $FD000-$FEFFF (blocks $FD and $FE). The following system Call would map them to your address space:

```
ldb #2          number of blocks
ldx #$FD        starting block number
os9 F$MapBlk    call MapBlk
stu IOPorts     save address where mapped
```

Upon return, the U register will have the starting address where the blocks were switched in. For example, suppose the system call above returned $4000. To access an I/O port at extended address $FD020 you would write to logical address $4020.

Other system calls that copy data to or from one task's map to another are available (F$STABX, F $MOVE, etc.). Some of these are made system mode privileged in some systems, usually depending on the type of applications the system is designed for. They can be unprotected if desired by changing the appropriate bit in the corresponding entry of the system service request table and making a new system boot with the patched table.

3.5. Overview of Multiprogramming

OS-9 is a multiprogramming operating system, which allows several independent programs (called “processes” or “tasks”) to 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 called “ticks” that are typically 100 milliseconds in duration. Processes that are “active” (meaning not waiting for some event) are run for a specific system-assigned period called a “time slice”. How often a process receives a time slice depends on a process' priority value relative to the priority of all other active processes. Many OS-9 service requests are available to create, terminate, and control processes.

3.6. 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. The creation process is outlined as follows:

1. 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.

2. 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 in Level I, and 512 bytes in Level II, 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 is not permitted to access the descriptor. See os9sysdefs (P$'s) for information on what is in a process descriptor.

3. 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 “fork” system call requested an optionally larger' size. OS-9 then attempts to allocate a memory area of this size from the free memory space.

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

The new process is also assigned a unique number called a ("process ID" which is used as its identifier. Other processes can communicate with it by referring to its ID in various system calls. The process also has 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 “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.

3.7. Process States

At any instant, a process can be in one of four 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.

SUSPENDED - The process is in the active queue but is awaiting I/O completion. The implementation of suspend state is in the device drivers and its use is optional.

There is a queue for each process state (except suspend). 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.

3.7.1. The Active State

The active 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 (described in section 3-8) that gives all active processes some CPU time, even if they have a very low relative priority.

3.7.2. The Wait State

Wait 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.

3.7.3. The Sleeping State

Sleep 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.

3.8. 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.

3.9. Signals

“Signals” are an asynchronous control mechanism used for inter-process 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. Interrupts are masked when inside the intercept routine, so the intercept routine should be as short as possible.

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.

3.10. 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.

3.10.1. 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	LEVEL ONE ADDRESS	LEVEL TWO ADDRESS
SWI3	$002C	$00F2
SWI2	$002E	$00F4
FIRQ	$0030	$00F6
IRQ	$0032	$00F8
SWI	$0034	$00FA
NMI	$0036	$00FC

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 must 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. The following tables 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	
SWI2	$FFE4	
FIRQ	$FFE6	
IRQ	$FFE8	
SWI	$FFEA	
NMI	$FFEC	
WARM	$FFEE	Reserved for warm-start

VECTOR	ADDRESS
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.

3.10.2. 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-generating 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 pointer 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 privileged 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 the use of a “module directory”. The module directory contains the address of each module and a count of how many processes are using each particular module. When modules are loaded into memory, they are added to the module directory. When a process links to a module in memory, its link count is incremented by one. When modules are no longer needed (a link count of 0), their memory is deallocated and the module name removed from the directory (except ROMS, which are discussed later). In many respects, modules and memory in general, are managed 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 and 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 operating systems.

4.1. 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 $800FE3.

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

4.2. 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

4.2.1. Type/Language Byte

The module type is coded into the four 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:

Code	Name	Meaning
$10	Prgrm	Program module
$20	Sbrtn	Subroutine module
$30	Multi	Multi-module (for future use)
$40	Data	Data module
$50-$B0	User-definable	
$C0	Systm	OS-9 System module
$D0	FlMgr	OS-9 File Manager module
$E0	Drivr	OS-9 Device Driver module
$F0	Devic	OS-9 Device Descriptor module

Note

0 is not a legal type code.

The four least significant bits of byte 6 describe the language type as listed below:

0	Data	Data (non-executable)
1	Objct	6809 object code
2	ICode	BASIC09 I-code
3	PCode	PASCAL P-code
	The following are currently not implemented:	
4	CCode	C I-code
5	CblCode	COBOL I-code
6	FrtnCode	FORTRAN I-code

The purpose of the language type is so high-level language run-time systems can verify that a module is of the correct type before execution is attempted. BASIC09, for example, may run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

4.2.2. Attribute/Revision Byte

The upper four bits of this byte are reserved for module attributes. Currently, only bit 7 is defined, and when set indicates the module is reentrant and therefore “sharable”.

The lower four bits are a revision level from zero (lowest) to fifteen. If more than one module has the same name, type, language, etc., OS-9 only keeps in the module directory the module having the highest revision level. This is how ROMed modules can be replaced or patched: you load a new equivalent module having a higher revision level. Because all modules locate each other by using the LINK system call which searches the module directory by name, it always returns the latest revision of the module, wherever it may be.

NOTE: A previously linked module can not be replaced until all processes which linked to it have unlinked it (after its link count goes to zero). When a new module is loaded, it will go into the module directory. Both copies will be in memory, but previous users will use the old copy until they unlink from the old and link to the new module.

4.2.3. Other Level II Memory Management Characteristics

The following features apply only to Level II systems.

Preloading Modules

Memory modules loaded by the LOAD command or F$LOAD system call are loaded into memory and added to the module directory but are not mapped into any process' address space until executed or LINKed. Thus, a large number of modules can be preloaded into memory for fast access when needed. By judicious use of LOAD, LINK and UNLINK more than 64K of modularized code can be accessed by one task.

Shared Data Modules

Because one or more memory modules (hence, memory blocks) can be mapped simultaneously into several logical maps (and frequently are for programs such as Basic09), RAM data modules can be used to allow two or more tasks to share a common data area. The assembler is used to create a memory module with the proper header, CRC, etc., and PCB, FDB, FCC, FCS, directives only to initialize all required data space.

Write Protect

If the system MMU has this feature and the OS-9 option is enabled, OS-9 will write-protect any blocks containing Sharable memory modules. Sharable modules have the “reentrant” bit set in the module header. This increases system security by protecting all tasks sharing a copy of a memory module from another process accidentally (or deliberately) altering the shared module. Note that in order to “patch” modules in systems with write protect the “reentrant” atribute bit must be cleared (on disk) prior to loading into memory.

4.3. Typed Module Headers

As mentioned before, the first nine bytes of the module header are defined identically for all module types. There is usually more header information immediately following, the layout and meaning varies depending on the specific module type. Module types $C - $F are used exclusively by OS-9. Their format is given elsewhere in this manual.

The module type illustrated below is the general purpose “user” format that is commonly user for OS-9 programs that are called using the FORK or CHAIN system calls. These modules are the “user-

defined" types having type codes of 5 through E. They have four more bytes in their headers defined as follows:

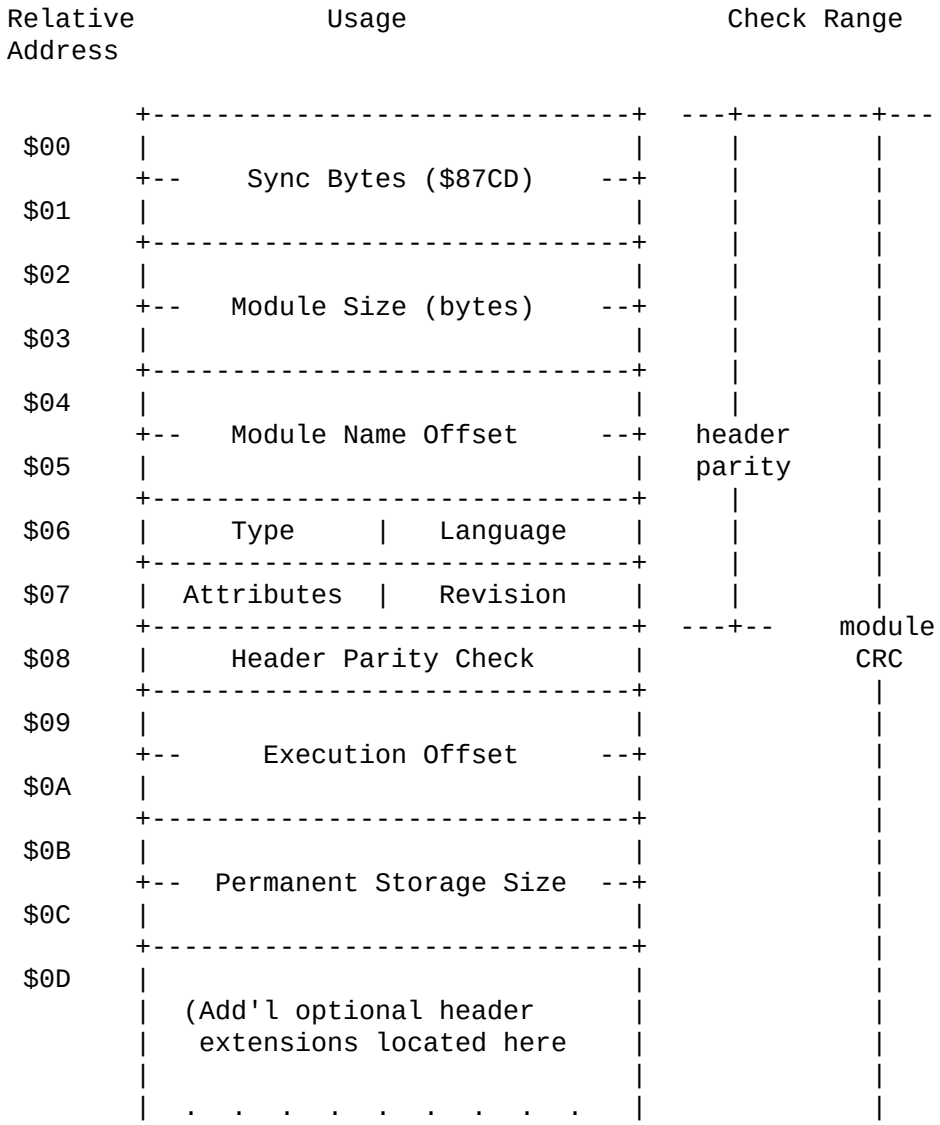
MODULE OFFSET **DESCRIPTION**

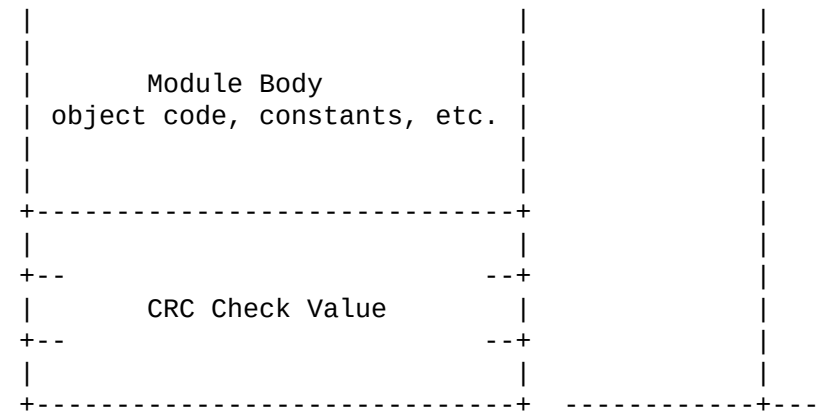
$9,$A = Execution Offset. The program or subroutine's starting address, relative to the first byte of the sync code. Modules having multiple entry points (cold start, warm start, etc.) may have a branch table starting at this address.

$B,$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.

4.4. Executable Memory Module Format





4.5. 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.

4.6. Memory Module Examples

The following examples show the structure of two OS-9 memory modules. The first is a typical terminal descriptor, and the second is a data module that is used to share data between processes.

Example 4.1. Terminal Device Descriptor

```
 nam TERM
 ttl Device Descriptor for terminal

 use defsfile

*************************
*  TERMINAL device module

 mod TrmEnd,TrmNam,DEVIC+OBJCT,REENT+1,TrmMgr,TrmDrv
 fcb UPDAT. mode
 fcb $F port bank
 fdb A.TERM port address
 fcb TrmNam-*-1 option byte count
 fcb DT.SCF Device Type: SCF
```

```
* DEFAULT PARAMETERS

 fcb 0 case=UPPER and lower
 fcb 1 backspace=BS,SP,BS
 fcb O delete=backspace over line
 fcb 1 auto echo on
 fcb 1 auto line feed on
 fcb 0 null count
 fcb 1 end of page pause on
 fcb 24 lines per page
 fcb C$BSP backspace char
 fcb C$DEL delete line char
 fcb C$CR end of record char
 fcb C$EOF end of file char
 fcb C$RPRT reprint line char
 fcb C$RPET dup last line char
 fcb C$PAUS pause char
 fcb C$INTR Keyboard Interrupt char
 fcb $11 Keyboard Quit char
 fcb C$BSP backspace echo char
 fcb C$BELL line overflow char
 fcb A.T.init ACIA initialization
 fcb 0 reserved
 fdb TrmNam offset of echo device   (continued)
 fcb 0 Transmit Enable char
 fcb 0 Transmit Disable char
TrmNam fcs "TERM" device name
TrmMgr fcs "SCF" file manager
TrmDrv fcs "ACIA" device driver

 emod Module CRC

TrmEnd EQU *
```

Example 4.2. Data Module

```
          nam  DataMod
          use  defsfile
LEVEL     equ  1 select level 1
          ifp1
          endc
TYPE      set  DATA      load value of $40
REVS      set  REENT+1   load value of $81
          mod  DataMEnd,DataMNam,TYPE,REVS,DataMEnt,0
DataMNam  fcs  "DataMod"
          fcb  0
DataMEnt  fcs  /12345678901234567890/
          fcs  /23456789012345678901/
          fcs  /34567890123456789012/
          fcs  /45678901234567890123/
          emod
DataMEnd  equ  *
```

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

OS-9 Level I and Level II provides a unified system-wide hardware independent I/O system for user programs and OS-9 itself. All I/O service requests (system calls) 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 then calls the file managers which in turn call the 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 and used while the system is running.

The structural organization of I/O related modules in an OS-9 system is hierarchical, as shown on page 2-1.

5.1. 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/to 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 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. The module to be linked to is the device's descriptor, which contains the names of the device driver and file manager for the device. The information in the device descriptor is saved by IOMAN so subsequent system calls can be routed to these modules.

5.2. 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. File managers are also responsible for mass storage allocation and directory processing if applicable to the class of devices they service.

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

The file managers are reentrant, and one file manager may be used for an entire class of devices having similar operational characteristics.

The three standard OS-9 file managers are:

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

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

PIPEMAN: Pipe File Manager which supports interprocess communication via “pipes”.

5.2.1. Anatomy Of a File Manager

Every file manager must have a branch table in exactly the following format. Routines that are not used by the file manager should branch to an error routine which sets the carry and loads B with an appropriate error code before returning. Routines returning' without error must insure the carry is clear.

```
* All routines are entered with:
*  (Y) Path Descriptor ptr
*  (U) Caller's register stack pointer

EntryPt equ *
        lbra Create
        lbra Open
        lbra MakDir
        lbra ChgDir
        lbra Delete
        lbra Seek
        lbra Read
        lbra Write
        lbra ReadLn
        lbra WriteLn
        lbra GetStat
        lbra PutStat
        lbra Close
```

Open, Create

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

Makdir

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

ChgDir

On multi-file devices, ChgDir searches for a file which must be a directory file. If the directory is found, the address of the directory (up to four bytes) are saved in the caller's process descriptor at P $DIO+2 (data directory) or P$DIO+8 (execution directory).

In the case of RBF, the address of the directory's file descriptor is saved. Open/Create begins searching in this directory when the caller's pathlist does not begin with a “/” character. File managers that do not support directories should return the carry set and an appropriate error code in (B).

Delete

Multi-file device managers usually do a directory search that is similar to Open and, once found, remove the file name from the directory. Any media that was in use by the file is returned to unused status. In the case of RBF, space is returned and marked as available in the free cluster bit map on the disk. File managers that do not support multi- file devices simply return an error.

Seek

File managers that support random access devices use Seek to position file pointers of the already open path to the byte specified. Typically, this is a logical movement. No error is produced at the time of the seek if the position is beyond the current “end of file”. File managers that do not support random access should do nothing. It is conceivable that an SCF-type manager could use seek to perform cursor positioning.

Read

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

Write

The Write request, like Read, must be capable of recording pure binary data without alteration. Usually, the routines for read and write are almost identical with the exception that Write uses the device driver's output routine instead of the input routine. RBF and similar random access devices that use fixed-length records (sectors) must often pre- read a sector before writing it unless the entire sector is being written. Writing past the end of file on a device should expand the file with new data.

ReadLn

ReadLn differs from Read in two respects. First, ReadLn is expected to terminate when the first end-of-line characater (carriage return) is encountered. ReadLn should also perform any input editing that is appropriate for the device. In the case of SCF, editing involves handling backspace, line deletion, removing the high-order bit from characters, etc.

WriteIn

Writeln is the counterpart of ReadLn. It should call the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing may also be performed. For example, SCF outputs a line feed and carriage return character and nulls if appropriate for the device, as well as pausing at the end of a screen page.

GetStat, PutStat

The GetStat (Get Status) and PutStat (Put Status) system calls are wild card calls designed to provide a method of accessing features of a device (or file manager) that are not generally device independent. The file manager may perform some specific function such as setting the size of a file to a given value. Status calls that are “unknown” should be passed on to the driver to provide a further means of device dependence. For example, a PutStat call to format a disk track may behave differently on different types of disk controllers.

Close

Close is responsible for insuring that any output to a device is completed (writing out the last buffer if necessary), and releasing any buffer space allocated in an open or create. It should not execute the device driver's terminate routine, but may do specific end-of-file processing if desired, such as writing end-of-file records on tapes or form feeds to printers.

5.2.2. Interfacing to the Device Driver

Strictly speaking, device drivers must conform to the general format presented in this manual. However, IOMAN uses only the “Init” and "Terminate" entry points. Other entry points need only be compatible with the file manager for which the driver is written. For example, the Read entry point of an SCF driver is expected to return one byte from the device. On the other hand, RBF expects Read to return an entire sector.

The following code is extracted from the SCF file manager to illustrate how a file manager might call one of it's drivers.

```
***************
*  IOEXEC
```

```
*    Execute Device's Read/Write routine

*  Passed: (A)=output char (write)
*          (X)=Device Table entry ptr
*          (Y)=Path Descriptor ptr
*          (U)=offset of routine (DSRead, D$Write)
* Returns: (A)=Input char (read)
*          (B)=error code, CC set if error
* Destroys B,CC

IOEXEC pshs a,x,y,u save registers
       ldu V$STAT,x get static storage for driver
       ldx V$DRIV,x get driver module address
       ldd M$EXEC,X and offset of execution entries
       addd 5,s offset by read/write
       leax d,x absolute entry address
       lda ,s+ restore char (for write)
       jsr 0,x execute driver read/write
       puls x,y,u,pc return (A)=char, (B)=error

       emod Module CRC
Size equ * size of Sequential File Manager
```

5.3. 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 controllers. 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 $E0). 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 descriptor” and the hardware control register address in the MPU registers. The branch table looks like:

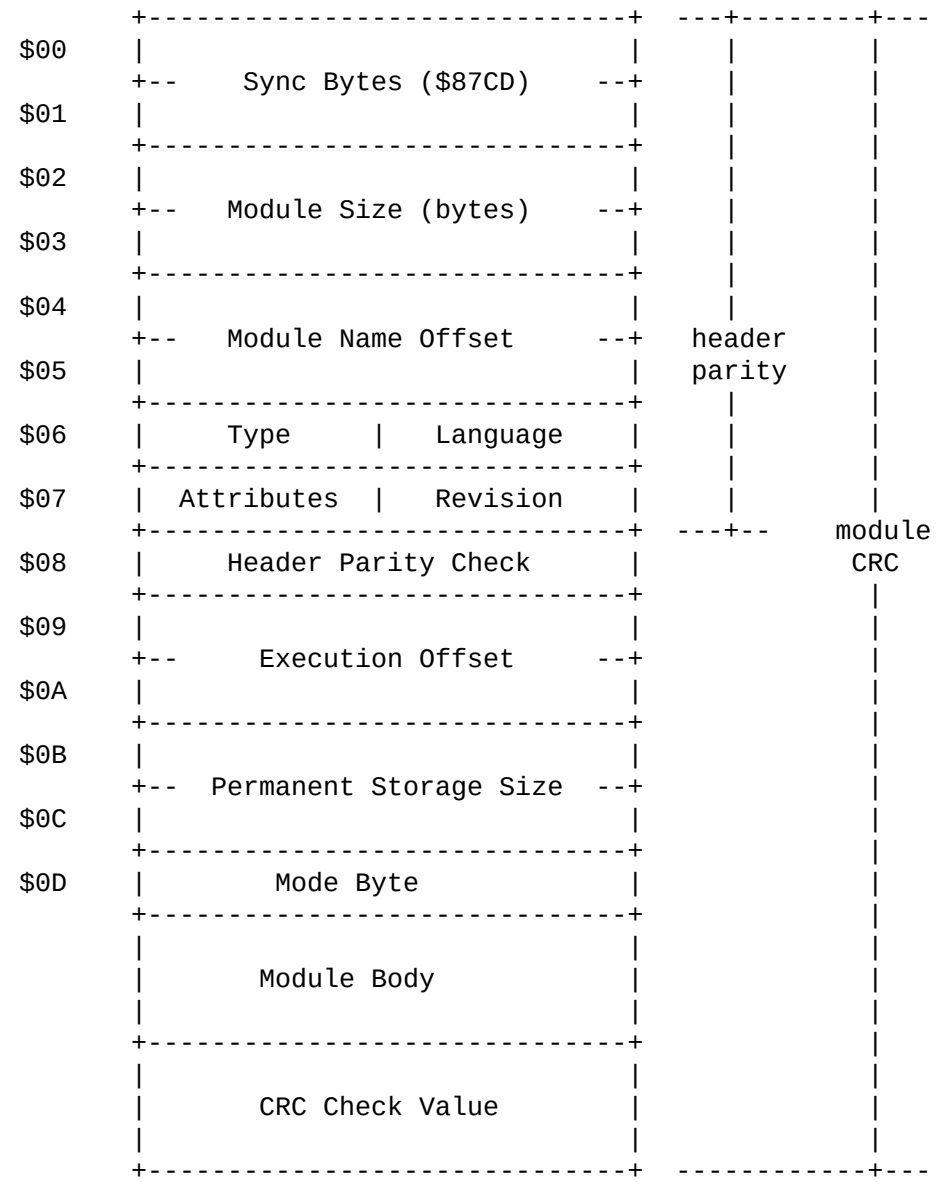
+0 = Device Initialization Routine
+3 = Read From Device
+6 = Write to Device
+9 = Get Device Status
+$C = Set Device Status
+$F = Device Termination Routine

For a complete description of the parameters passed to these subroutines see the file manager descriptions.

See the following page for a diagram of device driver format.

DEVICE DRIVER MODULE FORMAT

Relative Address	Usage	Check Range



$D Mode Byte - (D S PE PW PR E W R)

5.3.1. OS-9 Interacting with Real World Devices

Device drivers are often in the position of waiting for hardware to accomplish a task or waiting for a user to enter or receive data. These Situations can occur when an SCF device driver receives a read and no data is available, or when a write is received and there is no buffer space available. '

Any driver operating under OS-9 should release the current process from running (via F$Sleep) to allow other processes to continue using the CPU time when it encounters a conflict of the type described above. The most efficient way for the driver to come out of the sleep and resume processing data is by interrupts (IRQs). It is possible for the driver to sleep for a number of system clock ticks and then check the device or buffer for ready. The drawbacks to this technique are:

1. - It requires the system clock to always be active.

2. - It may take 2 clock ticks, or maybe even 20 ticks for the device to become ready, which leaves the programmer with a dilemma. If the programmer chooses to sleep for 2 ticks, he or she wastes CPU time awakening and checking for device ready. If the driver sleeps 20 ticks, it does not have good response time.

An interrupt system allows the hardware to report to the CPU and the device drivers when the device has finished some operation. Using interrupts to advantage, a device driver may set up interrupt handling to occur when a character is sent/reed, a disk operation is complete, or whatever. The OS-9 environment is set up for ease of interrupt processing. There is a flexible built-in polling system facility for pausing a process, and awakening the process. Microware has developed a technique for device drivers to follow in order to process interrupts.

Step 1: Init routine places the driver local IRQSVC routine in the IRQ polling sequence via an F$IRQ system call.

```
ldd V.Port,u get address to poll
leax IRQPOLL,pcr point to IRQ packet
leay IRQSERVC,pcr point to IRQ service routine
OS9 F$IRQ add dev to poll sequence
bcs Error abnormal exit if error
```

Step 2: Whenever a driver program must wait for the hardware, it should call a sleep routine. The sleep routine will copy V.Busy to V.Wake, then it will go to sleep for some period of time.

Step 3: When the driver program “awakens”, it will check whether it awakened because of an interrupt or a signal sent from some other process. The usual way to accomplish the check is with the V.Wake storage byte. The V.Busy byte is maintained by the file manager to be the process ID of the process using the driver. When V.Busy is copied into V.Wake, then V.Wake becomes a flag byte and an information byte. A non-zero Wake byte indicates there is a process awaiting an interrupt. The value in the Wake byte indicates what process should be awakened by the sending of a wakeup signal. The following code will indicate a technique to accomplish this:

```
          lda V.Busy,u get proc ID
          sta V.Wake,u arrange for wakeup
          andcc #^IntMasks clear the way for interrupts
Sleep50   ldx #0 or any tick time desired.
          OS9 F$Sleep await an IRQ
          ldx D.Proc get process desc ptr (if signal test)
          ldb PSSignal,x is signal present? (if signal test)
          bne SigTest bra if so (if signal test)
          tst V.Wake,u IRQ occur?
          bne SleepSO bra if not
```

Note that the code labelled “if signal test” is only necessary if the driver wishes to return to the caller if a signal is sent without waiting for the device to finish. Also note that IRQs (and FIRQs) must be masked between the time a command is given to the device and the moving of V.Busy to V.Wake. If they are not masked, it is possible for the device IRQ to occur and the IRQSERVC routine to become confused as to sending a wakeup signal or not.

Step 4: When the device issues an interrupt, the routine address given in the F$IRQ will be called. This routine is called as if it were a portion of the interrupt handler in the system. The interrupts are masked, the routine should be as short as possible, and the routine should return to the caller via RTS, since the system poller has called it via JSR and will do the RTI when done. The IRQSERVC routine may want to verify that an interrupt has occurred for the device. It will need to clear the interrupt and retrieve any data in the device. Then the V.Wake byte is used to communicate back to the main driver routine. If V.Wake is non-zero, it should be cleared (indicating a true device interrupt), and its contents used as the process ID for and F$Send system call sending a wakeup signal to the process. Some sample code follows:

```
          ldx V.Port,u get device address
          tst ???? is it real interrupt from this device?
          bne IRQSVC9O bra to error if not
```

```
          lda Data,x get data from device
          sta 0,y store data in buffer (simplified example)
          lda V.Wake,u get process ID
          beg IRQSVC8O bra if none
          clr V.Wake,u clear it as flag to main routine
          ldb #S$Wake get wakeup signal
          OS9 F$Send send signal to driver
IRQSVC8O  clrb clear the carry bit (this indicates all is well)
          rts
IRQSVC90  comb set the carry bit (this indicates bad IRQ call)
          rts
```

5.3.2. SUSPEND STATE - A New Feature for LII V1.2

With the advent of OS-9 Level II Version 1.2 there is a new possibility for device drivers when working with IRQs which involves the use of the suspend bit in the process state byte. The scheduler has been changed to ignore any process in the active queue which has the suspend state bit set. The main advantage of this method over the previous method is the elimination of the F$Send system call during the interrupt handling. Since the process is already in the active queue, it need not be moved from one queue to another. The device driver IRQSERVC routine can now clear the suspend bit in the process state in order to “wakeup” the suspended main driver. Sample routines for the sleep and IRQSERVC calls follow:

```
          lda D.Proc get process ptr
          sta V.Wake,u prep for re-awakening

          enable device to IRQ, give command, etc.

          bra Cmd50 enter suspend loop

   Cmd30  ldx D.Proc get ptr to process desc
          lda P$State,x get state flag
          ora #Suspend put proc in suspend state
          sta P$State,x save it in proc desc
          andcc #^IntMasks unmask interrupts
          ldx #1 give up time slice
          OS9 F$Sleep suspend (in active queue)
   Cmd50  orcc #IntMasks mask interrupts while changing state
          ldx D.Proc get proc desc addr (if signal test)
          lda P$Signal,x get signal (if signal test)
          beq SigProc bra if.signal to be handled
          lda V.Wake,u true interrupt?
          bne Cmd30 bra if not
          andcc #^IntMasks assure interrupts unmasked
```

Note D.Proc is a pointer to the process descriptor of the current process. These descriptors are always allocated on 256 byte page boundaries. Thus, having the high order byte of the address is adequate to locate the descriptor. D.Proc is put in V.Wake as a dual value, in one instance it is a flag byte indicating a process is indeed suspended, and in the other instance it is a pointer to the process descriptor so the IRQSERVC routine can clear the suspend bit. It is necessary to have the interrupts masked from the time the device is enabled until the suspend bit has been set to insure that the IRQSERVC routine will not think it has cleared the suspend bit before it is even set. Then when the bit is set, the process could go into permanent suspension. The IRQSERVC routine sample follows:

```
         ldy V.Port,u get dev addr
         tst V.Wake,u is process awaiting IRQ?
```

```
          beq IRQSVCER no exit

          clear device interrupt
          exit if IRQ not from this device

          lda V.Wake,u get process ptr
          clrb
          stb V.Wake,u clear proc waiting flag
          tfr d,x get process descriptor ptr
          lda F$State,x get state flag
          anda #^Suspend clear suspend state
          sta P$State,x save it
          clrb clear carry bit
          rts

IRQSVCER comb set carry bit
          rts
```

5.4. Device Descriptor Modules

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

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.

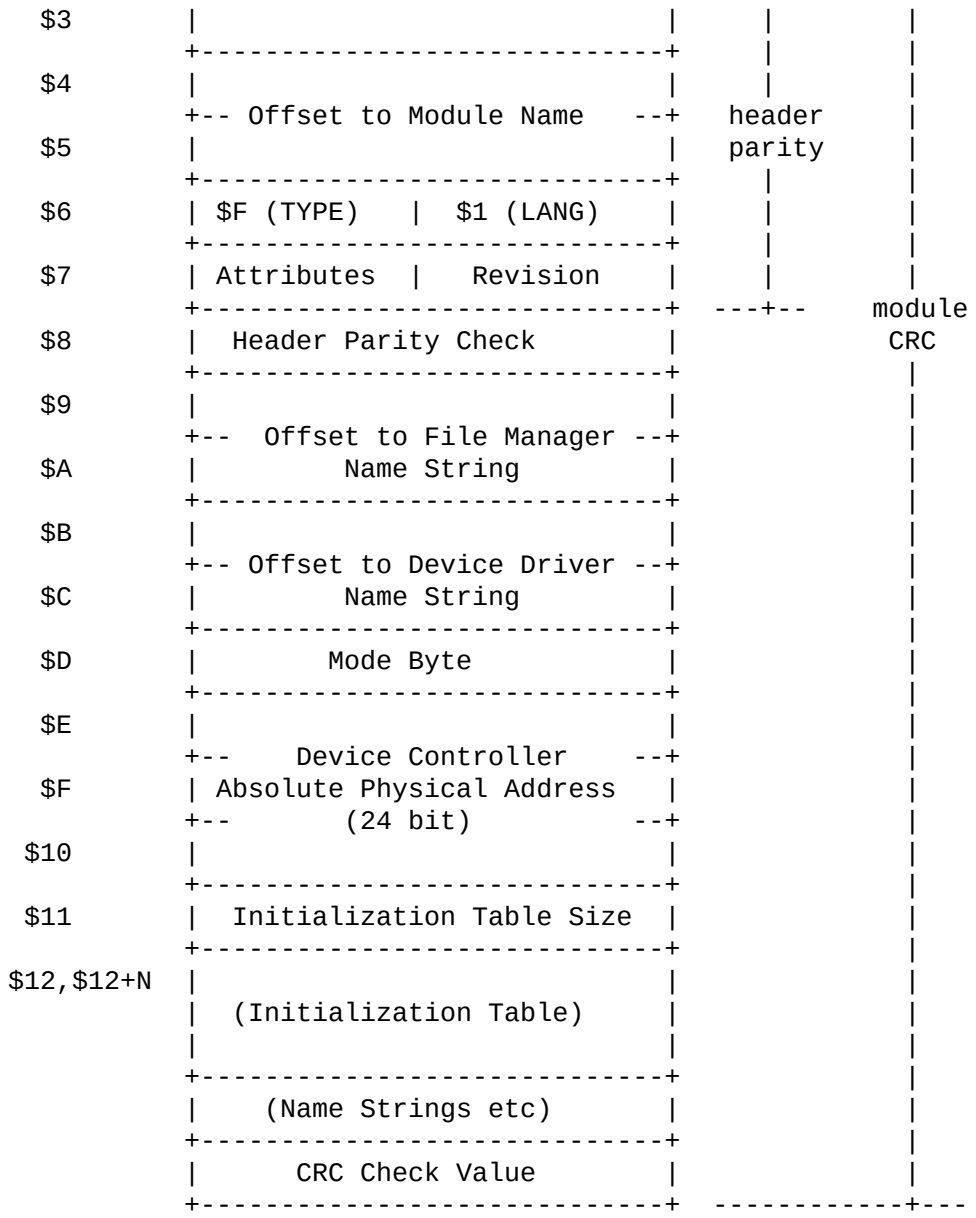
The name of the module is the name the device is known by to the system and user (i.e. it is the device name given in pathlists). Its format consists of a standard module header that has a type “device descriptor” (code $F). The rest of the device descriptor header consists of:

$9,$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 initialization 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.

```
MODULE     DEVICE DESCRIPTOR MODULE FORMAT
OFFSET

           +------------------------------+  ---+--------+---
  $0       |                              |     |        |
           +--   Sync Bytes ($87CD)    --+     |        |
  $1       |                              |     |        |
           +------------------------------+     |        |
  $2       |                              |     |        |
           +--   Module Size           --+     |        |
```



5.5. 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

Name	Addr	Size	Description
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.

“PD.OPT” and “PD.FST” sections are defined for each file manager in the assembly language equate file (OS9 `SCFDefs` for SCF and OS9 `RBFDefs` for RBF).

Chapter 6. Random Block File Manager

The Random Block File Manager (RBF) 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. RBP can handle any number or type of such systems simultaneously. RBF 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, RBF requests allocation and deallocation of 256 byte data buffers; usually one is required for each open file. When physical I/O functions are necessary, RBF directly calls the subroutines in the associated device drivers. All data transfers are performed using 256 byte data blocks. RBF 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 RBF 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.

6.1. 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, file structure, and storage allocation information.

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

Logical sector one (LSN 1) is the beginning of an allocation map which indicates which disk sectors are free and available for use in new or expanded files. The allocation bit map may be up to 256 sectors for high volume media.

The volume's root directory usually starts at logical sector two or it will start at the logical sector following the allocation map. Its logical sector number is given in the information LSN 0 (DD.Dir).

6.1.1. Identification Sector

Logical sector number zero contains a description of the physical and logical characteristics of the volume which 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	FD sector of root directory
DD.OWN	$0B	2	Owner's user number

Name	Addr	Size	Description
DD.ATT	$0D	1	Disk attributes
DD.DSK	$0E	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
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
DD.OPT	$3F	32	Path descriptor options

6.1.2. Disk Allocation Map

Logical sector one of the disk, and possibly more sectors, is used for the “disk allocation map” that specifies which clusters on the disk are available for allocation of file storage space. The size of the allocation map can be up to a maximum of 256 sectors decided by the Format utility. Format sets the size of the bitmap depending on disk size and sectors per cluster. DD.MAP specifies the number of bytes that 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. Multiple sector bitmaps allow the number of sectors per cluster to be as small as possible for high volume media. 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.

6.1.3. 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.Creat	$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 “nonsharable” 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 that specifies the beginning of the block, and a two byte block size (in sectors). The entry following the last segment must 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 (IT.SAS), or the number of sectors requested, whichever is 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 used wherever possible rather than adding a new segment. When the file is closed, unused sectors in the last segment are truncated and returned to the free pool.

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. Multiple segments can be avoided by writing a byte at the highest address to be used on a file before writing any other data.

6.1.4. 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 hold the name and LSN of a regular file 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.

6.2. RBF Definitions of the Path Descriptor

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

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
RBF 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
RBF 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)

Name	Addr	Size	Description
PD.STP	$22	1	Step rate
PD.TYP	$23	1	Device type
PD.DNS	$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 RBF and RBF-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 and 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.

6.3. RBF Device Descriptor Modules

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

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 RBF path descriptor)
$16	IT.DNS	RMB 1	media density (0 - SINGLE, 1-DOUBLE)
$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 1 = no verify

Module Offset			
$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.

6.4. 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 RBF. The device driver is free to use the remainder in any manner. 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)

OFFSET		ORG 0	
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
7		RMB 8	reserved
	DRVBEG	EQU .	beginning of drive tables
F	TABLES	RMB DRVMEM*N	reserve n drive tables
	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 contains the process ID of the last process to use the device. Not used by RBF-type device drivers.

V.BUSY contains the process ID of the process currently using the device. Defined by RBF.

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

TABLES This area contains one table for each drive that the controller will handle (RBF will assume that there are as many tables as indicated by V.NDRV). Some time after the driver INIT routine has been called, RBF 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	track size in sectors
$04	DD.MAP	RMB 2	# bytes in allocation map
$06	DD.BIT	RMB 2	number of sectors per bit (CLUSTER SIZE)
$08	DD.DIR	RMB 3	address of root directory
$0B	DD.OWN	RMB 2	owner user number
$0D	DD.ATT	RMB 1	disk attributes
$0E	DD.DSK	RMB 2	disk id
$10	DD.FMT	RMB 1	media format
$11	DD.SPT	RMB 2	sectors/track
$13	DD.RES	RMB 2	reserved for future use
	DD.SIZ	EQU .	

Offset		**ORG 0**	
$15	V.TRAK	RMB 2	current Track Number
$17	V.BMB	RMB 1	bit-map use flag
$18	V.FileHd	RMB 2	open file list for this drive
$1A	V.DiskID	RMB 2	disk id
$1C	V.BMapSz	RMB 1	bitmap size
$1D	V.MapSct	RMB 1	lowest reasonable bit map sector
$1E		RMB 8	reserved
$26	DRVMEM	EQU .	size of each drive table

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

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

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

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

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

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

DD.ATT 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 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 contains the current track which the head is on and is updated by the driver.

V.BMB is used by RBF 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.

6.5. RBF Device Drivers

As with all device drivers, RBF-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.

6.5.1. NAME: INIT

NAME:	INIT
INPUT:	(Y) = address of the device descriptor module (U) = address of device static storage
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C BIT SET (B) = 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. In Level Two the buffer can be declared in static storage.

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.

6.5.2. NAME: READ

NAME:	READ
INPUT:	
	(B) = msb of disk logical sector number (X) = lsb's of disk logical sector number (Y) = address of the path descriptor (U) = address of the device static storage
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.

6.5.3. NAME: WRITE

NAME:	WRITE
INPUT:	
	(B) = msb of disk logical sector number (X) = lsb's of disk logical sector number (Y) = address of the path descriptor (U) = address of the device static storage
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. It is recommended that the compare loop be as short as possible to keep the necessary sector interleave value to a minimum.

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.

6.5.4. NAME: GETSTA PUTSTA

NAME:	GETSTA/PUTSTA
INPUT:	(U) = address of the device static storage area (Y) = address of the path descriptor FUNCTION CODE SHOULD BE RETRIEVED FROM RSB ON THE USER STACK.
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	CONDITION 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

6.5.5. 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.

6.5.6. 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 RBF, it is a 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.

6.5.7. NAME: BOOT (Bootstrap Module)

NAME:	BOOT (Bootstrap Module)
INPUT:	None.
OUTPUT:	(D) = Size of the boot file (in bytes) (X) = Address of where the boot file was loaded in memory
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 Section 6.1.2, “Disk Allocation Map”.

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.

6.6. RBF Record Locking

Record locking is a general term that refers to mechanisms that are designed to preserve the integrity of files that can be accesses by more than one user or process. The OS-9 implementation of record locking is designed to be as invisible as possible to application programs so existing programs do not have to be rewritten to take advantage of record facilities, and most new programs may be written without special concern for multi-user activity.

Simply stated, record locking involves recognizing when a process is trying to read a record that is currently being modified by another process, and if the record is “locked out”, deferring the read until the record is “safe”. This scheme is referred to as conflict detection and prevention. RBF record locking also takes care of non-sharable file locking and deadlock detection.

6.6.1. Record Locking and Unlocking

Conflict detection must determine when a record is in the process of being updated. RBF edition 16 and above provides true record locking on a byte basis. Earlier versions of RBF locked out all sectors in the particular record's area. A typical record update sequence is:

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

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

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

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

6.6.2. Non-sharable Files

File Locking may be considered a special case of record locking, in which the entire file is considered unsafe to be used by more than one user. Sometimes (although rarely), it is necessary to create a file that can never be accessed by more than one user at a time by setting the non-sharable(S) bit in the file's attribute byte. The bit can be set by using an option when the file is created, or later using the Attr utility. Once the non-sharable bit has been set, only one user may open the file at a time. If other users attempt to open the file, an error (#253) will be returned.

More commonly, a file will need to be declared non-sharable only during the execution of a specific program by opening the file with the non-sharable bit set in the mode. An example would be when the file is being sorted. With the non-sharable bit set, the file will be treated exactly as though it had a non-sharable attribute. If the file has already been opened by another process, an error (#253) will be returned.

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

6.6.3. End of File Lock

A special case of record locking occurs when a user reads or writes data at the end of file. The user is said to have “EOF Lock” and will keep the end of file locked out until a read or write is performed that is not at the end of the file. EOF Lock is the only case that a write call automatically causes any of the file to be locked out. It was done to avoid problems that could otherwise occur when two users want to simultaneously extend a file.

An interesting and extremely useful side effect occurs when a program creates a file for sequential output. As soon as the file is created, EOF Lock is gained, and no other process will be able to “pass” the writer in processing the file. For example, if an assembly listing is redirected to a disk file, a spooler utility might open and begin listing the file before the assembler has -written even the first line of output. Record locking will always keep the spooler 'one step behind' the assembler, making the listing come out as desired.

6.6.4. DeadLock Detection

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

When a deadly embrace is found by RBF, a deadlock error (#254) is returned to the process that caused the deadlock to be detected. It is easy to create programs that, when ran together, generate lots of deadlock errors. The easiest way to avoid them is to access records of shared files in the same sequences in processes that may be run simultaneously.

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

6.6.5. Specific Details for Particular I/O Functions

Open/Create

The most important rule to follow when opening files is do not open a file for update if you only intend to read from it. Files open for read only will not cause records to be locked out, and they will generally

help the system to run faster. If files are routinely opened for update on a multi-user system, users may sometimes become record locked for extended periods of time, When this occurs, users sometimes think the system has died, and exhibit panic behavior.

File permission checking occurs for all files encountered in the specified pathlist. Permission checking means that if you do not have permission to read a directory, you may not access any files in that directory.

The special “@” file should be used in update mode with extreme care. To keep system overhead low, record locking routines only check for conflicts on paths opened for the same file. The “@” file is considered different from any other file, and therefore will only conform to record lockouts with other users of the “@” file. Writing via the “@” file (to patch crashed disks, for example) should only be done in single-user mode. The '@' file has been included as a convenience only; it likely that problems will eventually occur if it is used in update mode regularly.

Read/ReadLine

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

requested. Thus, if a ReadLine call is made for 256 bytes, exactly 256 bytes will be locked out, regardless of how many bytes are actually read before a carriage return is encountered. EOF Lock will be gained if the bytecount requested also includes the current end of file.

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

Write/WriteLine

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

Close

When RBF expands a file it expands it in increments of at least the 'segment allocation size' (IT.SAS in device descriptor) sectors long so that usually more space than is required is allocated. At the time the file is closed, the excess space is trimmed and returned to free space. This strategy does not work very well for random-access data bases that expand frequently by only a few records. The segment list rapidly fills up with small segments. A provision has been added to prevent this from being a problem.

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

Seek

The seek call has no effect on record locking with the minor exception noted above in close. In particular, seek does not remove any record locks.

Makdir

Makdir creates its files in non-sharable mode. Since file attributes are checked at each pathlist element (see open/create), makdir will return an error if it cannot gain non-sharable access to any directory

specified. It can be a bit annoying sometimes, but it helps prevent certain recursive programs from getting out of control.

Del

The delete begins by opening the file for write in non-sharable mode. If the file is open, an error (#253) is returned, and the file is NOT deleted. All sorts of problems occur when this is not enforced.

SetStatus

Two new setstat codes have been added for the convenience of record locking. They are SS.Lock, for locking or releasing part of a file; and SS.Ticks, for setting the length of time a program is willing to wait for a locked record. See the I$SETSTT documentation for a description of the codes.

Chapter 7. Sequential Character File Manager

The Sequential Character File Manager (SCF) is the OS-9 file manager module that supports devices that operate on a character-by-character basis, such as terminals, printers, modems, etc. SCF 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 SCF and two SCF-type device driver modules: ACIA, which run 6850 serial interfaces, and PIA, which drives a 6821-type parallel interface for printers.

7.1. SCF Line Editing Functions

I$Read and I$Write service requests to SCF-type devices (which correspond to Basic09 GET and PUT statements) pass data to/from the device without any modification. In particular, carriage returns are not automatically followed by line feeds or nulls, and the high order bits are passed as sent/received. If X-on and X-off are enabled, these characters are intercepted by the device driver.

I$ReadLn and I$WritLn service requests (which correspond to Basic09 INPUT, PRINT, READ and WRITE statements) to SCF-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 transferred 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. SCF will return an end-of-file error on I$Read or I$ReadLn if this is the first (and only) character input. It can be disabled by setting its value to zero.

If PD.RPR <> 0, SCF (I$ReadLn) will, upon receipt of this character, echo a carriage return (and insert it in the buffer for “DUP” described below), 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 (if any) with an error identical to the keyboard interrupt signal code. PD.INT 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 interrupt 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.

7.2. SCF Definitions of The Path Descriptor

The table below describes the path descriptors used by SCF and SCF-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
PD.FST	$0A	32	File Manager Storage
PD.OPT	$20	(size of option section)	Option section
SCF 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	Readline 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
SCF 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.XON	$38	1	ACIA X-ON char
PD.XOFF	$39	1	ACIA X-OFF char

The first section is universal for all file managers, the second and third section are specific for SCF and SCF-type device drivers. The option section of the path descriptor contains many device operating parameters which may be read or written by the OS9 I$GetStt or I$SetStt service requests. IOMAN initializes this section when a path is opened 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.

7.3. SCF Device Descriptor Modules

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

```
MODULE                  ORG $12
OFFSET
                TABLE   EQU .   beginning 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)
$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
```

MODULE OFFSET		ORG $12	
$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 namestring offset
$2A	IT.XON	RMB 1	xon character
$2B	IT.XOFF	RMB 1	xoff character
$2C	IT.STN	RMB 2	offset to status routine
$2E	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, if the end of file character (offset $13) was a zero, there would be no end of file character.

The initialization value (offset $26) is typically used to initialize the device's control register when a path is opened to it.

7.4. SCF Device Driver Storage Definitions

An SCF-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 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 driver. IOMAN determines that a new incarnation of the device driver is needed when an attach occurs for a device with a 'different port address. The size of the storage area is given in the device driver module header. Some of this storage area is required by IOMAN and SCF, the device driver is free to use the remainder for variables and buffers. This static storage is defined in OS9 `IODEFS` and OS9 `SCFDEFS` 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)
$5	V.WAKE	RMB 1	process id to reawaken

OFFSET		ORG 0	
	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.XON	RMB 1	X-on character
$10	V.XOFF	RMB 1	X-off character
$11	V.RSV	RMB 12	reserved
$1D	V.SCF	EQU .	end of scf definitions

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 received. Maintained by SCF.

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 SCF to prevent more than one process from using the device at the same moment. Defined by SCF.

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. Maintained 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 maintained by SCF 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 SCF.

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

V.DEV2 This location contains the address of the ECHO (attached) device's static storage area. Typically a device is its own echo device. However, it may not be, as in the case of a keyboard and a memory mapped video display. Maintained by SCF.

V.INTR Keyboard interrupt character. This is maintained by SCF, which copies its value from PD.INT in the path descriptor.

V.QUIT Keyboard abort character. This is maintained by SCF which copies its value from PD.QUT in the path descriptor.

V.PCHR Pause character. This is maintained by SCF 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 SCF calls one of the device driver routines.

7.5. SCF Device Driver Subroutines

As with all device drivers, SCF device drivers use a standard executable memory module format with a module type of “device driver” (CODE $E0). 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 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.

7.5.1. NAME: INIT

NAME:	INIT
INPUT:	(U) = address of device static storage (Y) = address of device descriptor module
OUTPUT:	NONE
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
FUNCTION:	INITIALIZE DEVICE AND ITS STATIC STORAGE

Usually this routine has three basic operations to do:

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 SCF.

7.5.2. 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 indefinately.

Later when data is received, 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 for queueing data between the main driver and the IRQ service routine are *not* automatically allocated. It any are used, they should be defined in the device's static storage area.

7.5.3. NAME: WRITE

NAME:	WRITE
INPUT:	(A) = char to write (Y) = address of the path descriptor (U) = address of device static storage
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 sends 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 for queueing data between the main driver and the IRQ service routine are *not* automatically allocated. It any are used, they should be defined in the device's static storage area.

7.5.4. NAME: GETSTA/SETSTA

NAME:	GETSTA SETSTA
INPUT:	(A) = function code (Y) = address of path descriptor (U) = address of device static storage
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. Most SCF-type requests are handled by IOMAN or SCF. Any codes not defined by them will be passed to the device driver.

In writing getstat/setstat codes, it may be necessary to examine or change the register stack which contains the values of the 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. Note that Y is a pointer to the path descriptor and PD.RGS is the offset 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	condition code register

OFFSET	MNEMONIC			MPU 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

Sample access:

```
ldx PD.RGS,y
ldd R$Y,x
```

gets the Y register parameter from the caller

7.5.5. 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 use count in the device table becomes zero. In Level One systems, the termination routine is not called until the link count of the driver, descriptor, or file manager also reaches zero, and the module is being removed from the system memory directory. 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: LI - Modules contained in the BOOT file will NOT be terminated. LII - Any I/O devices that are not being used will be terminated.

7.5.6. 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 SCF, it is an important routine in device drivers. The main things that it does are:

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

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 empty, disable the device's “ready to transmit” interrupts.

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

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

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

Chapter 8. The Pipe File Manager

The Pipe File Manager (Pipeman) handles control of processes that use paths to pipes. Pipes allow concurrently executing processes to communicate data by allowing the output of one process (the writer) to be read as input to a second process (the reader). The writer sends it's output to standard output which is usually the terminal screen. The reader read input from standard input. When the “!” operator is used, Pipeman handles reading and writing to the pipe. Pipeman allocates a path descriptor and a 256 byte data buffer that the processes will read/write to/from. Pipeman also controls which process has centrol of the pipe (either a reader or a writer). See the *OS-9 Operating System User's Manual* for more information on pipes.

Pipeman has the standard file manager branch table at its entry point:

```
PipeEnt  lbra Create
         lbra Open
         lbra MakDir
         lbra ChgDir
         lbra Delete
         lbra Seek
         lbra PRead
         lbra PWrite
         lbra PRdLn
         lbra PWrLn
         lbra Getstat
         lbra Putstat
         lbra Close
```

For pipes, the MakDir, ChgDir, Delete, and Seek are illegal service routines and will return E$UnkSvc (unknown service request). Getstat and Putstat are “no action service routines” and will return with no error.

Create and Open are the same routine. They set up the 256 byte data buffer and save several addresses in the path descriptor.

Close checks to see if any process is reading or writing through the pipe. If not, the buffer is returned.

PRead, PRdLn, PWrite, PWrLn read/write data to/from the buffer.

The “!” operator tells shell that processes wish to communicate via a pipe. Example: OS9: proc1 ! proc2. In this example, shell will fork proc1 with the stdout path to a pipe and will fork proc2 with the stdin path from a pipe. Shell can also handle a series of processes using pipes, such as proc1 ! proc2 ! proc3 ! proc4. The outline on the next page shows how to set up pipes between two process.

8.1. Outlines of Establishing a Pipe Between Two Processes in a Machine Language Program

Example 8.1. Example: Establishing a Pipe Between Two Processes

```
Open /pipe  save path in variable x
Dup path #1 save stdout in variable y
Close #1    make path available
Dup x       puts pipe in stdout (Dup uses lowest avail.)
```

```
Fork proc1  fork process 1
Close #1    make path available
Dup y       restores stdout
Close y     make path available
Dup path #0 save stdin in y
Close #0    make path available
Dup x       puts pipe in stdin
Fork 2      fork process 2
Close #0    make path available
Dup y       restore stdin
Close x     no longer needed
Close y     no longer needed
```

The following example shows how an application program could spawn another process with the stdin and stdout of the process routed to a pipe.

Example 8.2. Example 2: Forking a process with standard paths to the pipe.

```
Open /pipe1  save path in variable a
Open /pipe2  save path in variable b
Dup 0        save stdin in variable x
Dup 1        save stdout in variable y
Close 0      make path available
Close 1      make path available
Dup a        make pipel stdin
Dup b        make pipe2 stdout
Fork new process
Close 0      make path available
Close 1      make path available
Dup x        restore stdin
Dup y        restore stdout
Return a&b   return pipe path numbers to caller
```

Chapter 9. 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). OS-9 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.

9.1. 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

9.2. 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.

9.3. 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.

9.4. 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.

9.5. 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 shown 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.

9.6. 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.

9.7. A Sample Program

The OS-9 **list** utility and “Inkey” program is shown on this and the following pages as an example of assembly language programming.

Example 9.1. List Utility

```
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 87CD004E           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   *
```

Example 9.2. Basic09 InKey Subroutine

```
***************
* INKEY - a subroutine for BASIC09
*    Author: Robert Doggett

* Calling syntax:
*    RUN InKey(StrVar)
*    RUN InKey(Path,StrVar)

*    Inkey determines if a key has been typed on the given path
* (Standard Input if not specified), and if so, returns the next
* character in the string variable.  If no key has been
* typed a null string is returned.  If a path is specified, it may
* be either type BYTE or INTEGER.  StrVar may be declared as a
* BYTE variable, if preferred. If this is done, a value of 255
* indicates that no data is ready.

  0038            E$Param  equ   56     Basic09's "Parameter Error"
  0021             TYPE     set   SBRTN+OBJCT
  0081             REVS     set   REENT+1
  0000 87CD005F             mod   InKeyEnd,InKeyNam,TYPE,
                                  REVS,InKeyEnt,0
  000D 496E6B65    InKeyNam fcs   "Inkey"
  0012 02                   fcb   2          edition two

D 0000                      org   0          Parameters
D 0000             Return   rmb   2          Return addr of caller
D 0002             PCount   rmb   2          Number of params
D 0004             Param1   rmb   2          after 1st param addr
D 0006             Length1  rmb   2          size
D 0008             Param2   rmb   2          2nd param addr
D 000A             Length2  rmb   2          size

  0013 3064        InKeyEnt leax  Param1,S
  0015 EC62                 ldd   PCount,S   Get parameter count
  0017 10830001             cmpd  #1         just one parameter?
  001B 2717                 beq   InKey20    ..Yes; path (A)=0
  001F 10830002             cmpd  #2         Two parameters?
  0021 2635                 bne   ParamErr   No, abort
```

```
0023 ECF804              ldd   [Param1,S] Get path number
0026 AE66                ldx   Length1,S
0028 301F                leax  -1,X       byte available?
002A 2706                beq   InKey10    ..Yes; (A)=Path number
002C 301F                leax  -1,X       Integer?
002E 2628                bne   ParamErr   ..No; abort
0030 1F98                tfr   B,A
0032 3068        InKey10 leax  Param2,S
0034 EE02        InKey20 ldu   2,X        length of string
0036 AE84                ldx   0,X        addr of string
0038 C6FF                ldb   #$FF
003A E784                stb   0,X        Init to null str
003C 11830002            cmpu  #2         Two-byte string?
0040 2502                blo   InKey30    ..No
0042 E701                stb   1,X       put terminator in 2nd b
0044 C601        InKey30 ldb   #SS.Ready
0046 103F8D              OS9   I$GetStt   Is any data ready?
0049 2508                bcs   InKey90    ..No; exit
004B 108E0001            ldy   #1
004F 103F89              OS9   I$Read     Read one byte
0052 39                  rts             return error status

0053 C1F6        InKey90 cmpb  #E$NotRdy
0055 2603                bne   InKeyErr
0057 39                  rts             (carry clear)

0058 C638        ParamErr ldb  #E$Param   Parameter Error
005A 43          InKeyErr coma
005B 39                  rts

005C 70F3D5              emod
005F             InKeyEnd equ  *

00000 error(s)
00000 warning(s)
$005F 00095 program bytes generated
$000C 00012 data bytes allocated
$2410 09232 bytes used for symbols
```

Chapter 10. Adapting OS-9 to a New System

10.1. Adapting OS-9 Level I to a New System

Thanks to OS-9's modular structure, OS-9 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 to be 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.

Microware sells source code for standard I/O drivers, and a “User Source Code Package” (on OS-9 format disk only) which contains source code to the 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.

Warning

Standard OS-9 software packages are licensed for use on a single system. OS-9 cannot be resold or otherwise distributed (even if modified) without a license. Contact Microware for information regarding software licenses.

10.2. 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, RBF, SCF, SHELL, and utilities, seldom require modification.

10.3. Using OS-9 in ROM-based Systems

One of OS-9's major features is its ability to reside in ROM 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 SCF, 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 RBF, and appropriate disk driver and device descriptor modules. As in (c) above, change device addresses and initialization if needed. If RBF *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.

10.4. 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., RBF 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.

10.5. 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 /                  (Module name to fork to)
fcc /userprg/                (Parameter to pass)
fcb $0D                      (expects carriage return terminator)

fcs /Basic09 /               (Start in Basic09)
```

Chapter 11. 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 processing 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.

NOTE: The system call descriptions that follow are explained using a particular notation. When F.xxx appears in the SYSTEM CALLS section, it means that a `BSR` or `LBSR` is made instead of a system call. The SYSTEM CALLS section shows the other routines the service request calls. Any notation followed by “*” means that no error checking is done on return from the system call. The DATA section shows what direct page information is accessed by the service request.

Thus, if a system call returns an error code, the user can trace its origin. Some system calls generate errors themselves; these are listed as POSSIBLE ERRORS. If the returned error code does not match any of the given possible errors, then it was probably returned by another system call made by the main call.

E$UnkSvc (Unknown Service Request) can be returned from any OS-9 system call to signal that the service request has not been installed.

11.1. User Mode Service Requests

11.1.1. F$AllBit - Set bits in an allocation bit map

ASSEMBLER CALL: OS9 F$AllBit

MACHINE CODE: 103F 13

ROUTINE LOCATION: LI - OS9p1
LII - OS9p2

INPUT: (D) = Base address of allocation bit map.
(X) = Bit number of first bit to set.
(Y) = Bit count (number of bits to set)

OUTPUT: Bits set in given allocation map.

ERROR OUTPUT: None.

FUNCTION: ALLBIT is used by OS-9 to maintain internal allocation maps. System memory is allocated using the following technique:

1. F$SchBit is called to locate free area.

2. F$AllBit is called to reserve or allocate the bits.

3. F$DelBit is called to release the bits when they are no longer needed.

The ALLBIT service request sets bits in the allocation bit map with the number of the bit to be set Specified by the X register. Each bit is equivalent to a certain amount of system resource be it memory, disk space, etc.

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

DATA: LI and LII - D.Proc

SYSTEM CALLS: LI - None.
LII - F$LDABX*, F$STABX*

CAVEATS: LI - Beware calling AllBit with Y = 0 (Bit count of zero!)

11.1.2. F$Chain - Load and execute a new primary module.

ASSEMBLER CALL: OS9 F$Chain

MACHINE CODE: 103F 05

INPUT: (A) = Language / type code.

(B) = Optional data area size (256 byte pages).
(X) = Address of module name or file name.
(Y) = Parameter area size (256 byte pages).
(U) = Beginning address of parameter area.

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

POSSIBLE ERRORS: LI - E$NEMod - module not executable.
E$DelSP - memory size of zero.
E$IForkP - not enough memory for stack and parameters.
LII - E$NEMod

FUNCTION: CHAIN is used when it is necessary to execute an entirely new program, but without the overhead of creating a new process. It is functionally similar to a FORK followed by an EXIT, but with less processing overhead.

The CHAIN 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.

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

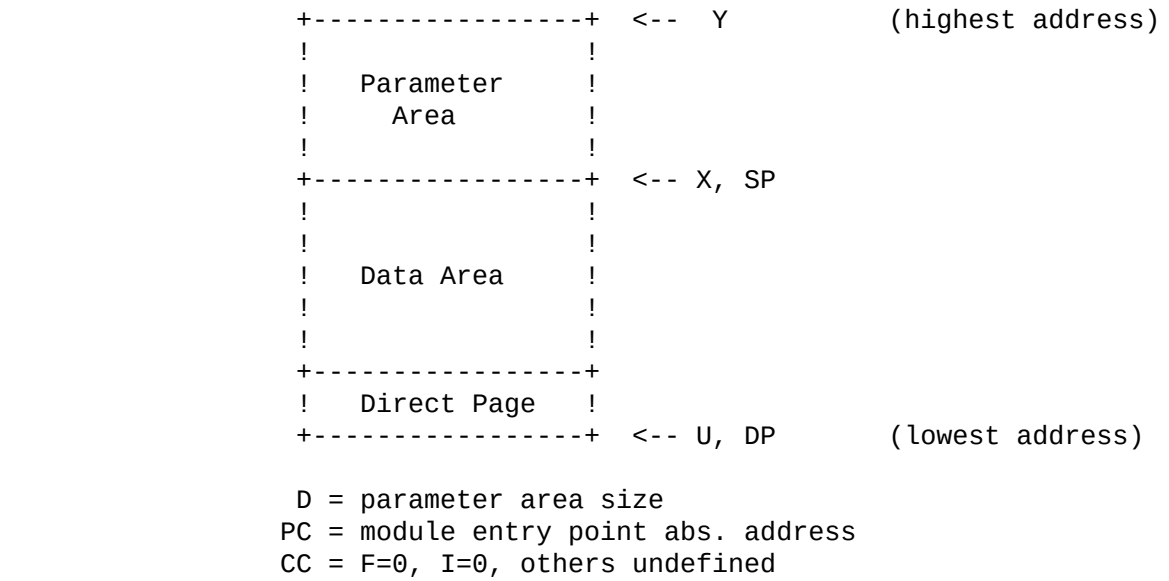
1. The process' old primary module is *unlinked*.

2. The system parses the name string of the new process' “primary module” - the program that will initially be executed. 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).

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

4. Intercepts and any pending signals are erased.

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



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”.

DATA: LI - D.Proc, D.Usrsvc
LII - D.Proc, D.Systsk, D.Sysprc, D.PrcDBT

SYSTEM CALLS: LI - F$UnLink*, F$Link, F$Load, F$Mem
LII - F.Allproc, F$UnLink*, F$SLink*, F$Load, F$LDDDXY*, FMem, FAllTsk*, F$LDABX*, F$STABX*, F$Move*, F$DelTsk*, F$SrtMem*, F$Aproc*, F$NProc*, F$Exit*

CAVEATS: LI and LII - Beware of chaining to system object module.
LII - Beware of memory size of zero in module header.

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

11.1.3. F$CmpNam - Compare two names

ASSEMBLER CALL: OS9 F$CmpNam

MACHINE CODE: 103F 11

ROUTINE LOCATION: LI - OS9p1
LII - OS9p2

INPUT: (B) = Length of first name.
(X) = Address of first name.
(Y) = Address of second name.

OUTPUT: (CC) = C bit clear if the strings match.

ERROR OUTPUT: None. CC indicates only the match/nomatch condition. B doesn't contain an error code.

FUNCTION: To be used in combination with parsename.

Given the address and length of a string, and the address of a second string, CMPNAM compares them and indicates whether they match.

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

DATA: LI - None.
LII - D.Proc, D.SysDAT

SYSTEM CALLS: LI - None.
LII - None.

CAVEATS: The second string must be terminated with the high order bit set.

11.1.4. F$CRC - Compute CRC

ASSEMBLER CALL: OS9 F$CRC

MACHINE CODE: 103F 17

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: (X) = Starting byte address.
(Y) = Byte count.
(U) = Address of 3 byte CRC accumulator.

OUTPUT: CRC accumulator is updated.

ERROR OUTPUT: None.

FUNCTION: To allow the system to easily generate/check CRC values of modules.

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

When checking an existing module CRC, the calculation should be performed on the entire module (including the module CRC). The CRC accumulator will contain the CRC constant bytes if .the module CRC is correct. Checking an existing CRC can also be done similar to below for checking a CRC match.

If the CRC of a new module is to be generated, the CRC is accumulated over the module (excluding CRC). The accumulated CRC is complemented then stored in the correct position in the module.

DATA: LI - None.
LII - D.Proc, D.SysTsk

SYSTEM CALLS: LI - None.
LII - None.

CAVEATS: CAVEATS: Be sure to initialize CRC accumulator only once for each module checked.

11.1.5. F$DelBit - Deallocate in a bit map

ASSEMBLER CALL: OS9 F$DELBIT

MACHINE CODE: 103F 14

ROUTINE LOCATION: LI - OS9p1
LII - OS9p2

INPUT: (D) = Bit number of first bit to clear.
(X) = Base address of an allocation bit map.
(Y) = Bit count (number of bits to clear).

OUTPUT: Bits cleared in allocation map.

FUNCTION: DELBIT is used by the system to maintain internal allocation maps. See also F$AllBit.

DELBIT 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.

DATA: LI - D.Proc
LII - None.

SYSTEM CALLS: LI - None.
LII - F$LDABX*, F$STABX*

CAVEATS: Beware of calling with Y = 0 (Bit count of zero).

11.1.6. F$Exit - Terminate the calling process.

ASSEMBLER CALL: OS9 F$EXIT

MACHINE CODE: 103F 06

ROUTINE LOCATION: LI - OS9p2
LII - OS9p2

INPUT: (B) = Status code to be returned to the parent process.

OUTPUT: Process is terminated.

ERROR OUTPUT: None.

FUNCTION: The F$EXIT 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 WAIT call, which returns to the parent the status byte passed by the child in its EXIT call. The status byte can be an OS-9 error code that the terminating process wishes to pass back to its parent process (the shell assumes this), or it 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.

The following information describes the order of operation of an F$EXIT call.

1. Close all paths.
2. Return memory to system.
3. Unlink primary module.
4. LII - Free task number. LI - Clear all sibling links and their parent ID. Free process descriptor of any dead child.
5. LII - Clear all sibling links and their parent ID. LI - If parent is dead, free the process descriptor.
6. If parent is alive: A. Search wait queue of parent.
    1. If parent cannot be found, note the process death in process state and leave the process in limbo until parent notices the death.
    2. If parent found then move parent to active queue, inform parent of death/status, remove child from sibling list, and free process descriptor for the system.

DATA: LI - D.Proc, D.PrcDBT, D.WProcQ
LII - D.Proc, D.SysStk, D.PrcDBT

SYSTEM CALLS: LI - I$Close*, F$SrtMem*, F$UnLink*, F$Find64*, F$Ret64*, F$AProc*
LII - I$Close*, F$DelImg*, F$UnLink*, F$DelTsk*, F.GetProc*, F.RetProc*

CAVEATS: Only the primary module is unlinked. Any module that is loaded or linked to by the process should be unlinked before calling F$EXIT.

11.1.7. F$Fork - Create a new process

ASSEMBLER CALL: OS9 F$Fork

MACHINE CODE: 103F 03

ROUTINE LOCATION:	LI - OS9p1 LII - OS9p2
INPUT:	(A) = Language / Type code. (B) = Optional data area size (pages). (X) = Address of module name or file name. (Y) = Parameter area size. (Number of bytes) (U) = Beginning address of the parameter area.
OUTPUT:	(X) = Updated past the name string. (A) = New process ID number.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	LI - E$PrcFul, E$NEMod, E$DelSP, E$IForkP LII - E$NEMod, E$PrcFul

FUNCTION: FORK creates a new process which becomes a “child” of the caller, and sets up the new process' memory, MPU registers, and standard I/O paths.

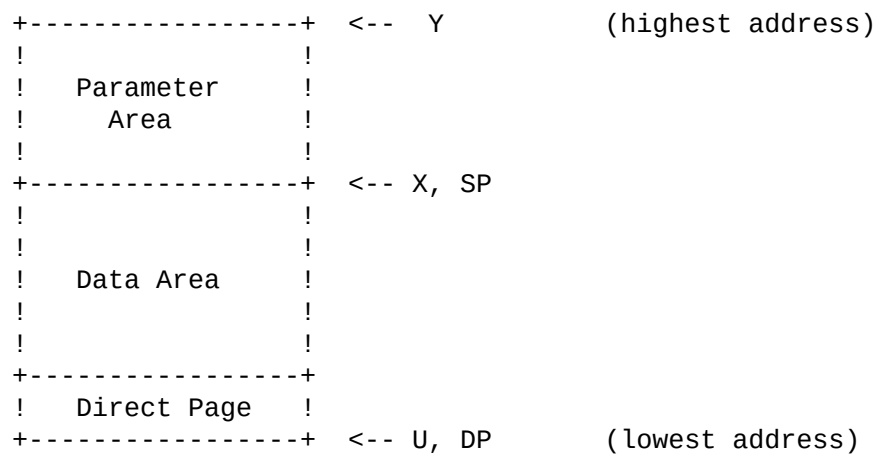
The system parses the name string of the new process' “primary module” — the program that will initially be executed. Then the system module directory is searched to see if the program is already in memory. If so, the module is linked to and executed. If not, the name string is used as the pathlist of the file which is to be 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 RAM area equal to the required data storage size or the passed parameter optional data area size whichever is larger, (includes the parameter passing area, which is copied from the parent process' data area). In LI the RAM area must be contiguous, and in LII the RAM areas are pieced together. 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.



```
 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. Thus, anything started from shell will always have a parameter area ending with a carriage return.

DATA: LI - D.PrcDBT, D.Proc, D.UsrVC
LII - D.Proc, D.SysTsk, D.PrcDBT

SYSTEM CALLS: LI - F$ALL64, I$Dup*, F$Aproc*, F$EXIT, F$Link, F$Load, F$Mem.
LII - F.AllPrc, I$Dup*, F$SLink, F$Load, F$Mem, F$AllTsk*, F$Move*, F$DelTsk, F$AProc*, F$DelImg*, I$Close*, F$UnLink*, F$SRTMem, F $SRqMem.

CAVEATS: 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” until the process table becomes full. Also, beware of forking a process with a mem size of zero.

11.1.8. F$ICPT - Set up a signal intercept trap

ASSEMBLER CALL: OS9 F$ICPT

MACHINE CODE: 103F 09

ROUTINE LOCATION: LI - OS9p2
LII - OS9p2

INPUT: (X) = Address of the intercept routine.
(U) = Address of the intercept routine local storage.

OUTPUT: Signals sent to the process will cause the intercept routine to be called instead of the process being killed.

ERROR OUTPUT: None.

FUNCTION: ICPT 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 and keyboard interrupt.

The intercept routine is entered asynchronously because a signal may be sent at any time (similar to 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. Therefore, all ICPT routines should index off the U register.

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 ICPT routine is entered when a process is beginning a new time slice and it has a signal pending or is returning from a system call or an IRQ.

When the intercept conditions occur, a second stack frame is built below the normal program stack area. The second stack frame has a PC for the ICPT routine. Then an RTI will execute the ICPT routine. The interrupt masks are set during the ICPT routine so it should be short and fast.

After the ICPT routine has been completed, it should execute an RTI which will begin normal process execution.

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.

DATA: LI and LII - D.Proc

SYSTEM CALLS: None.

11.1.9. F$ID - Get process ID / user ID

ASSEMBLER CALL: OS9 F$ID

MACHINE CODE: 103F 0C

ROUTINE LOCATION: LI - OS9p2
LII - OS9p2

INPUT: None

OUTPUT: (A) = Process ID.
(Y) = User ID.

ERROR OUTPUT: None.

FUNCTION: 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.

DATA: LI - D.Proc
LII - D.Proc

SYSTEM CALLS: None.

11.1.10. F$Link - Link to memory module

ASSEMBLER CALL: OS9 F$LINK

MACHINE CODE: 103F 00

ROUTINE LOCATION: LI - OS9p1
LII - OS9p2

INPUT: (A) = Module type / language byte.
(X) = Address of the module name string.

OUTPUT: (A) = Module type / language.
(B) = Module attributes / revision level.
(X) = Advanced past the module name.
(Y) = Module entry point absolute address.

(U) = Module header absolute address.

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

POSSIBLE ERRORS: LI - EMNF, EModBsy
LII - EMNF, EModBsy, E$MemFul

FUNCTION: LINK causes OS-9 to search the module directory for a module having a name, language, and type as given in the parameters. If found, the address of the module's header is returned in 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 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.

In Level II if a module is part of a group of modules and a link has caused the entire group to be linked into local memory, the new link will not cause the group to be linked in again. OS9 will discover that the module is already available and the single module's link count will be incremented.

DATA: LI - D.ModDir, D.ModDir+2
LII - D.Proc, D.ModDir, D.ModEnd

SYSTEM CALLS: LI - F.PrsNam*, F.CNAM*
LII - F.PrsNam, F.CNAM*, F.LDDDXY*, F.FreeHb*, F.SetImg*, F.DattoLog<<*

11.1.11. F$Load - Load module(s) from a file

ASSEMBLER CALL: OS9 F$LOAD

MACHINE CODE: 103F 01

ROUTINE LOCATION: LI and LII - IOMAN

INPUT: (A) = Language / type (0 = any language / type)
(X) = Address of pathlist (file name)

OUTPUT: (A) Language / type
(B) = Attributes / revision level
(X) = Advanced past pathlist
(Y) = Primary module entry point address
(U) = Address of module header

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

POSSIBLE ERRORS: LI - E$BMID
LII - E$MemFul, E$BMID, E$MNF

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

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

DATA: LI - None.
LII - D.Proc, D.SysTsk, D.BlkMap, D.ModDir, D.ModEnd

SYSTEM CALLS: LI - I$Open, I$Read, F$SrqMem, F$VModul, F$SrtMem*, I$Close*
LII - F$Allprc, I$Open, F$Alltsk, F$Settsk, I$Read, F$Move*, F$VModul, I$Close*, F$DelPrc*, F$LDDDXY*, F$ELink

11.1.12. F$Mem - Resize data memory area

ASSEMBLER CALL: OS9 F$MEM

MACHINE CODE: 103F 07

ROUTINE LOCATION: LI and LII - OS9p2

INPUT: (D) = Desired new memory size in bytes.

OUTPUT: (Y) = Address of upper bound of new memory area.
(D) = Actual size of new memory area in bytes.

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

POSSIBLE ERRORS: LI and LII - E$DelSP, E$MemFul

FUNCTION: MEM is used to contract or expand 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 old highest address. If D = O, the call is taken to be an information request and the current upper bound and size will be returned.

This request can never return all of a process' memory, or the page in which its SP register points to. D must equal at least one to change memory size.

In Level One systems, the request may return an error upon an expansion request even though adequate free memory exits because the data area must always be contiguous, and memory requests by other processes may fragment memory into smaller, scattered blocks that are not adjacent to the caller's present data area. 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.

DATA: LI - D.Proc, D.PMBM
LII - D.Proc

SYSTEM CALLS: LI - F$SchBit, F$AllBit*, F$DelBit*
LII - F$AllImg, F$DelImg

11.1.13. F$PErr - Print error message

ASSEMBLER CALL: OS9 F$PErr

MACHINE CODE: 103F 0F

ROUTINE LOCATION: LI and LII - IOMAN

INPUT: (B) = Error code.

OUTPUT: Error Message.

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

FUNCTION: PERR is the system's error reporting utility. It writes an error message to the standard error path. Most OS-9 systems will display:

```
ERROR #<decimal number>
```

by default. The error reporting routine is vectored and can be replaced by a more elaborate reporting module. In Level 1 systems the reporting module can be replaced. In Level 2 systems, a system routine may replace the module, but the replacement will affect all users. Microware has not implemented any PERR modification on Level 2 systems.

DATA: LI - D.Proc
LII - D.Proc, D.Systsk

SYSTEM CALLS: LI - I$Writln*
LII - F$Move*, I$Writln*

11.1.14. F$PrsNam - Parse a path name

ASSEMBLER CALL: OS9 F$PrsNam

MACHINE CODE: 103F 10

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: (X) = Address of the pathlist.

OUTPUT: (X) = Updated past the optional “/”
(Y) = Address of the last character of the name +1
(A) = Trailing byte (Delimiter character)
(B) = Count of characters found.

ERROR OUTPUT: (CC) = C bit set.
(B) = Appropriate error code.
(Y) = Address of first nondelimeter char in string

POSSIBLE ERRORS: LI and LII - E$BNam

FUNCTION: PrsNam 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. PrsNam 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 Y will be moved past any space or comma characters so the next pathlist in a command can be parsed.

NOTE: PrsNam processes only one name, so several calls may be needed to process a pathlist that has more than one name. PrsNam will terminate a name on recognizing a delimiter character or high order bit set. It will skip one trailing comma or any number of trailing spaces.

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
```

DATA: LI - None

SYSTEM CALLS: LI - None
LII - F.DATLog*, P.LDAXY*

11.1.15. F$SchBit - Search bit map for a free area

ASSEMBLER CALL: OS9 F$SchBit

MACHINE CODE: 103F 12

ROUTINE LOCATION: LI - OS9p1
LII - OS9p2

INPUT: (D) = Beginning bit number
(X) = Beginning address of bit map.
(Y) = Bit count (free bit block size)
(U) = End of bit map address.

OUTPUT: (D) = Beginning bit number.
(Y) = Bit count.

ERROR OUTPUT: (CC) = C bit set.
D and Y are returned, but Y will be less than specified.

FUNCTION: SCHBIT searches the specified allocation bit map starting at the beginning bit number (D) 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.

DATA: LI - None
LII - D.Proc, D.Systsk

SYSTEM CALLS: LI - None
LII - F$LDABX*

11.1.16. F$Send - Send a signal to another process

ASSEMBLER CALL: OS9 F$SEND

MACHINE CODE: 103F 08

ROUTINE LOCATION: LI and LII - OS9p2

INPUT: (A) = Receiver's process ID number.
(B) = Signal code.

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

POSSIBLE ERRORS: LI and LII - E$IPrcID, E$USigP

FUNCTION: SEND sends a signal to the process specified. The signal code is a single byte value 0 - 255.

If the destination process for the signal is sleeping or waiting, it will be activated so that it may process the signal. The signal processing 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 code becomes

the exit status (see WAIT). An exception is the WAKEUP signal, which activates a sleeping process but does not cause the signal intercept routine to be examined and will not abort a process that has not run an F$ICPT..

Some of the signal codes have meanings defined by convention:

0 = System abort (unconditional)
1 = Wake up process
2 = Keyboard abort
3 = Keyboard interrup
128 - 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 try resending the signal later. It is good practice to issue a sleep call for a few ticks before a retry to avoid wasting CPU time.

NOTE: A process may send the same signal to multiple processes of the same ID by passing 0 as the receiver's process ID number. The superuser (ID number 0) may send the same signal to all processes by the same technique.

DATA:	LI - D.Proc, D.PrcDBT, D.SProcQ, D.WProcQ LII - D.Proc, D.SProcQ, D.WProcQ
SYSTEM CALLS:	LI - F$Find64, F$AProc* LII - F.GProcP, F$AProc*
CAVEATS:	Notice that the super user is capable of mass murder, by sending signal 0 to process 0.

11.1.17. F$Sleep - Put calling process to sleep

ASSEMBLER CALL:	OS9 F$Sleep
MACHINE CODE:	103F 0A
ROUTINE LOCATION:	LI and LII - OS9p2
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.

FUNCTION: SLEEP deactivates the calling process for a specified time, or indefinately if X = O. The process will be activated before the full time interval if a signal is received, therefore. sleeping indefinately is a good way to wait for a signal or interrupt without wasting CPU time.

The duration of a “tick” is system independent but is usually 100 milliseconds on Level I (50 ms with 6840) and 10 milliseconds on Level II.

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 to time more more than + or -1 tick. 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 X - 1 ticks occur and will resume execution when it reaches the front of the queue.

DATA:	LI - D.Proc, D.SProcQ

LII - D.Proc, D.SProcQ, D.Systsk

SYSTEM CALLS: LI - F$AProc*, F$NProc*
LII - F$AProc*, F$DelTsk*, F$NProc*

11.1.18. 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.

POSSIBLE ERRORS: LI and LII - E$IPrcID

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

DATA: LI - D.ProcDBT, D.Proc
LII - D.Proc

SYSTEM CALLS: LI - F$Find64
LII - F.GProcP

11.1.19. F$SSVC - Install function request

ASSEMBLER CALL: OS9 F$SSVC

MACHINE CODE: 103F 32

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: (Y) = Address of service request initialization table.

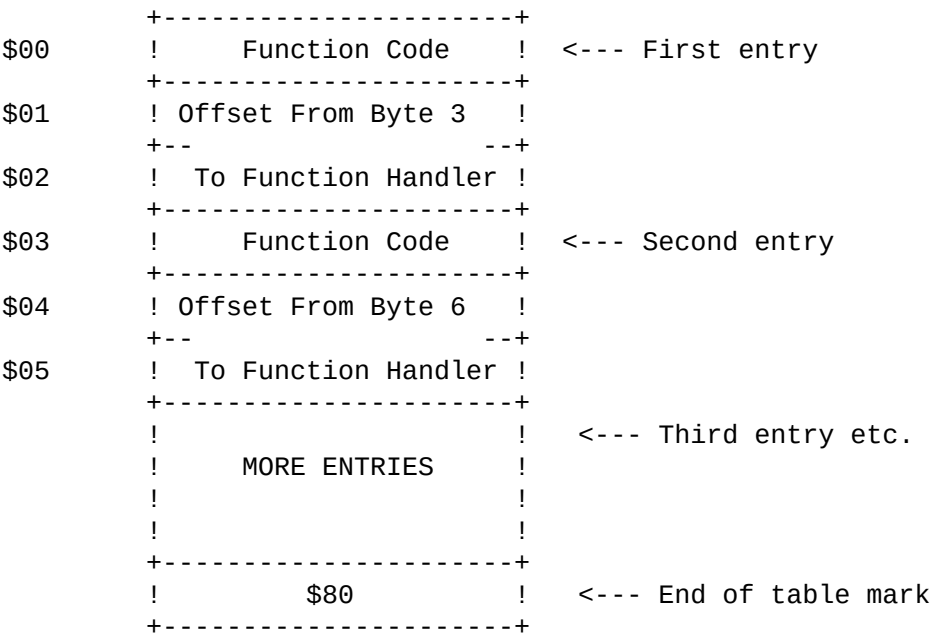
OUTPUT: None.

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

POSSIBLE ERRORS: LI - E$ISWI
LII - None.

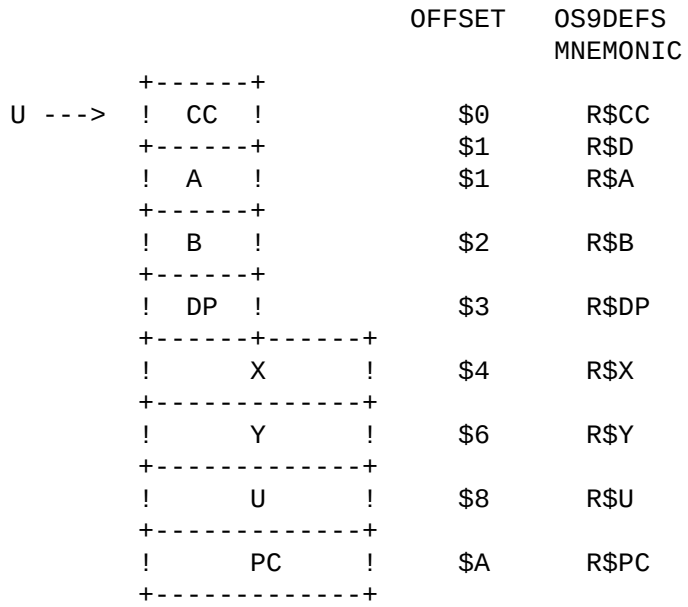
FUNCTION: SSVC is used to add a new function request to OS-9's user and privileged system service request tables, or to replace an old one. The Y register passes the address of a table which contains the function codes and offsets to the corresponding service request handler routines. This table has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

```
OFFSET
```



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 routine should process the service request and return from subroutine with an RTS instruction. They may alter any CPU 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:



NOTE: The user service routine should set the CPU registers CC and B to the appropriate values and return with RTS.- The service dispatcher will then set R$CC and R$B in the user's register stack.

LI - Function request codes are in the range 0 - $37

LII - Function request codes are in the range 0 - $7E

LI and LII function codes in the range $20 - $27 will not be used by Microware and are free for user definition.

DATA: LI and LII - D.SysDis
LII ONLY - D.UsrDis

CAVEATS: LII F$SSVC is only available to be called from system state by system addressed object code.

11.1.20. F$SSWI - Set SWI vector

ASSEMBLER CALL: OS9 F$SSWI

MACHINE CODE: 103F 0E

ROUTINE LOCATION: LI and LII - OS9p2

INPUT: (A) = SWI type code.
(X) = Address of user SWI service routine.

OUTPUT: None.

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

POSSIBLE ERRORS: LI and LII - E$ISWI

FUNCTION: SSWI sets up the interrupt vectors for SWI, SWI2 and SWI3 instructions. Each process has its own local vectors. Each SETSWI 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.

CAVEATS: Microware-supplied software 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.

DATA: LI and LII - D.Proc

SYSTEM CALLS: None

11.1.21. F$STime - Set system date and time

ASSEMBLER CALL: OS9 F$STIME

MACHINE CODE: 103F 16

ROUTINE LOCATION: LI and LII - OS9p2

INPUT: (X) = Address of time packet (see below)

OUTPUT: Time/date is set.

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

FUNCTION: STIME is used to set the current system date/time and start the system real-time clock. STIME is accomplished by putting the date/time packet in the system direct storage area, and then a

link system call is made to find the clock module. The clock initialization routine is called if the link is successful, and it is the duty of the clock initialization to:

1. Set up any hardware dependent functions (including moving new date/time into hardware if needed).

2. Set up the F$Time system call via F$SSVC.

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

DATA: LI - D.Day, D.Year, D.Min
LII - D.Time, D.Proc, D.SysPrc

SYSTEM CALLS: LI - F$Link
LII - F$Move*, F$Link

11.1.22. F$Time - Get system date and time

ASSEMBLER CALL: OS9 F$TIME

MACHINE CODE: 103F 15

ROUTINE LOCATION: LI and LII - Clock Module

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.

FUNCTION: TIME 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
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

DATA: Hardware dependent

CAVEATS: F$TIME is part of the clock module and will not exist if no previous call to F$STime has been made.

11.1.23. F$UnLink - Unlink a module

ASSEMBLER CALL:	OS9 F$UNLINK
MACHINE CODE:	103F 02
ROUTINE LOCATION:	LI - OS9p1 LII - OS9p2
INPUT:	(U) = Address of the module header.
OUTPUT:	None
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.

FUNCTION: UNLINK 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.itstemory 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. Also, any module in the OS9 Boot file cannot be deleted.

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

DATA:	LI - D.ModDir, D.BtLo, D.PmBm LII - None.
SYSTEM CALLS:	LI - F$IoDel, F$DelBit* LII - F$LDDDXY*, F$IoDel
CAVEATS:	If a bad address is passed, UnLink will NOT find a module in the module directory and will not return an error.

11.1.24. F$Wait - Wait for child process to die

ASSEMBLER CALL:	OS9 F$Wait
MACHINE CODE:	103F 04
ROUTINE LOCATION:	LI and LII - OS9p2
INPUT:	None
OUTPUT:	(A) = Deceased child process' process ID. (B) = Child process' exit status code.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	LI and LII - E$NoChld

FUNCTION: The calling process is deactivated until a child process terminates by executing an 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 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.

DATA: LI - D.Proc, D.PrcDBT, D.WProcQ
LII - D.Proc, D.WProcQ, D.PrcDBT, D.SysTsk

SYSTEM CALLS: LI - F$Find64*
LII - F.GProcP*, F$DelTSk, F$SrtMem*, F$NProc

CAVEATS: If the wait call returns with the carry bit set, then the wait was not performed. If the wait returns with the carry clear, then the wait functioned normally and any error that occurred in the child process will be returned in (B).

11.2. System Mode Service Requests

11.2.1. F$All64 - Allocate a 64 byte memory block

ASSEMBLER CALL: OS9 F$ALL64

MACHINE CODE: 103F 30

ROUTINE LOCATION: LI and LII - OS9p2

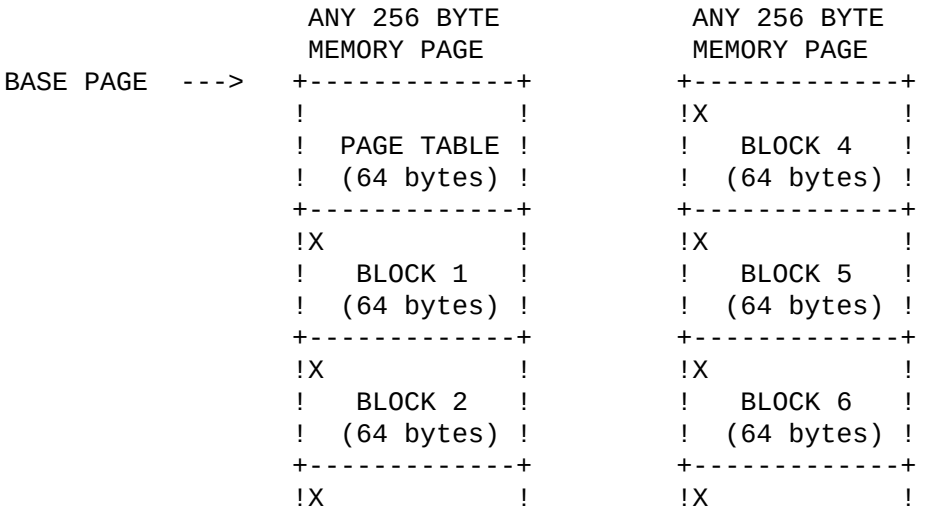
INPUT: (X) = Base address of page table (zero if the page table has not yet been allocated).

OUTPUT: (A) = Block number.
(X) = Base address of page table.
(Y) = Address of block.

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

POSSIBLE ERRORS: LI - E$PthFul

FUNCTION: ALL64 is used to dynamically allocate 64 byte blocks of memory by splitting whole pages (256 byte) into four 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:



In LI, ALL64 is used by OS-9 to allocate path descriptors and process descriptors. In LII, ALL64 is only used to allocate path descriptors because process descriptors for LII are 512 bytes.

DATA: LI and LII - None.

SYSTEM CALLS: LI and LII - F$SrqMem

Note

This is a privileged system mode service request.

11.2.2. F$AProc - Insert process in active process queue

ASSEMBLER CALL: OS9 F$APROC

MACHINE CODE: 103F 2C

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: (X) = Address of process descriptor.

OUTPUT: None.

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

FUNCTION: APROC inserts a process into the active process queue so that it may be scheduled for execution.

All processes already in the active process queue are aged, and the age of the specified process is set to its priority. The process is then inserted according to its relative age.

DATA: LI and LII - D.AProc

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

11.2.3. F$Find64 - Find a 64 byte memory block

ASSEMBLER CALL: OS9 F$FIND64

MACHINE CODE: 103F 2F

ROUTINE LOCATION: LI and LII - OS9p2

INPUT: (A) = Block number.
(X) = Address of base page.

OUTPUT: (Y) Address of block.

ERROR OUTPUT: (CC) = C bit set.

Indicates block not allocated or not in use.

FUNCTION: FIND64 will return the address of a 64 byte memory block as described in the F$ALL64 service request. OS-9 uses this service request to find prodess descriptors and path descriptors when-given their nUmber.

Block numbers range from 1..N

In LI, FIND64 is used to find process descriptors and path descriptors. In LII, it is used only to find path descriptors because process descriptors are 512 bytes in LII (see F$GProcP).

DATA: None.

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

11.2.4. F$IODel - Delete I/O device from system

ASSEMBLER CALL: OS9 F$IODEL

MACHINE CODE: 103F 33

ROUTINE LOCATION: LI and LII - I/O

INPUT: (X) = Address of an I/O module. (see description)

OUTPUT: None.

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

POSSIBLE ERRORS: LI and LII - E$ModBsy

FUNCTION: The X register passes the address of an I/O 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. IODEL is called by UNLINK when an I/O module is unlinked. I/O modules are device drivers, device descriptors, and file Managers.

LI - If a device is being unlinked for the final time (there are no other processes using the device), then IODEL performs the device termination routine. For LII this has been moved to the DETACH system call.

LII - IODEL returns information to the UNLINK system call after determining if a device is busy or not.

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

DATA: LI and LII - D.Init, D.DevTbl
LII - D.Proc, D.PrcDbt

SYSTEM CALLS: LI - F$SrtMem, F$Send, F$Find64, Makes call to driver terminate routine.
LII - None.

Note

This is a privileged system mode service request.

11.2.5. F$IOQu - Enter I/O queue

ASSEMBLER CALL: OS9 F$IOQU

MACHINE CODE: 103F ZB

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Process Number.

OUTPUT: None.

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

FUNCTION: IOQU links the calling process into the I/O queue of the specified process and performs an untimed sleep. It is assumed that routines associated with the specified process will send a wakeup signal to the calling process. IOQU is used primarily and extensively by IOMAN and file managers.

DATA: LI - D.Proc, D.PrcDBT
LII - D.Proc

SYSTEM CALLS: LI - F$Find64, F$Send*, F$Sleep*
LII - F$GProcP, F$Sleep*, F$Send*

Note

This is a privileged system mode service request.

11.2.6. F$IRQ - Add or remove device from IRQ table

ASSEMBLER CALL: OS9 F$IRQ

MACHINE CODE: 103F 2A

ROUTINE LOCATION: LI and LII - IOMAN

INPUT: (D) = Address of the device status register.
(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
(Y) = Device IRQ service routine address.
(U) = Address of service routine's static storage area.

OUTPUT: None.

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

POSSIBLE ERRORS: LI and LII - E$Poll. Error can be from full polling table or from bad mask byte (0).

FUNCTION: IRQ 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)= Add: 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 low 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

DATA: LI and LII - D.Init, D.PolTbl

Note

This is a privileged system mode service request.

11.2.7. F$NProc - Start next process

ASSEMBLER CALL: OS9 F$NProc

MACHINE CODE: 103F 2D

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: None.

OUTPUT: Control does not return to caller.

FUNCTION: 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.

DATA: LI - D.Proc, D.AProcQ, D.SWI2, D.UsrIRQ, D.SchIRQ
LII - D.SysPrc, D.Proc, D.SysTsk, D.AProcQ, D.TSlice, D.Slice, D.UsrSvc, D.XSWI2, D.UsrIRQ, D.XIRQ

SYSTEM CALLS: LI - F$EXIT*
LII - F$EXIT*, F.AProc*, F.AllTsk*

CAVEATS: The process calling NProc must already be in one of the three process queues. If it is not, then it will become unknown to the system even though the process descriptor still exists and will be printed out by a Procs command.

Note

This is a privileged system mode service request.

11.2.8. F$Ret64 - Deallocate a 64 byte memory block

ASSEMBLER CALL: OS9 F$Ret64

MACHINE CODE: 103F 31

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: (A) = Block number.
(X) = Address of the base page.

OUTPUT: None.

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

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

LI - F$Ret64 is used to free up or return a path descriptor or a process descriptor

LII - F$Ret64 is used to free only path descriptors.

DATA: LI and LII - None.

SYSTEM CALLS: F$SRTMem

Note

This is a privileged system mode service request.

11.2.9. 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.
(D) = New Memory size

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

POSSIBLE ERRORS: LI and LII - E$MemFul

FUNCTION: 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.

LII - Allocates memory for system address space only.

DATA: LI - D.FmBm
LII - D.SysMem, D.SysDAT, D.BlkMap, D.SysPrc

SYSTEM CALLS: LI - F$AllBit*
LII - F.AllImg

Note

This is a privileged system mode service request.

11.2.10. F$SRTMem - Return System Memory

ASSEMBLER CALL: OS9 F$SRTMem

MACHINE CODE: 103F 29

ROUTINE LOCATION: OS9p1

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.

POSSIBLE ERRORS: LI and LII - E$BPAddr

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

LII - Deallocates memory for system address space only.

DATA: LI - D.FmBm
LII - D.SysMem, D.SysDAT, D.BlkMap

SYSTEM CALLS: LI - F.DBit
LII - None.

Note

This is a privileged system mode service request.

11.2.11. F$VModul - Verify module

ASSEMBLER CALL: OS9 F$VMODUL

MACHINE CODE: 103F 2B

ROUTINE LOCATION: LI and LII - OS9p1

INPUT: LI - (X) = Address of new module
LII - (D) = DAT image pointer
(X) = New module block offset

OUTPUT: (U) = Address of module directory entry.

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

POSSIBLE ERRORS: LI and LII - E$KwnMod, E$DirFul, E$BMID, E$BMCRC
LII ONLY - E$BMHP

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

DATA: LI - D.FmBm, D.BtLo
LII - D.ModDir, D.ModEnd, D.BlkMap, D.ModDAT

SYSTEM CALLS: LI - F$DelBit*, F.FModul
LII - F.LDDDXY*, F.FModul, F$GCMDir*, F.LDAXY*, F$Sleep

Note

This is a privileged system mode service request.

11.3. I/O Service Requests

11.3.1. I$Attach - Attach a new device to the system.

ASSEMBLER CALL: OS9 I$ATTACH

MACHINE CODE:	103F 80
ROUTINE LOCATION:	LI and LII - IOMAN
INPUT:	(A) = Access mode. (X) = Address of device name string.
OUTPUT:	(U) = Address of device table entry. (X) = Updated past device name
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	LI - E$DevOvf, E$BMode, E$DevBsy LII - E$DevOvf, E$BMode, E$MemFul

FUNCTION: ATTACH 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. If 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 storage 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 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 (see Note below). Most devices are automatically installed, so it is used mostly when devices are dynamically installed or to verify the existence of a device. IOMAN attaches all devices at open, and detaches them at close.

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.

Note: Attach and Detach are a like Link and Unlink for devices, and they are usually used together. However, system performance can be improved slightly if all devices are attached at startup. This increments each device's use count and prevents the device from being reinitialized every time it is opened. This also has the advantage of allocating the static storage for devices all at once, which prevents fragmentation on Level One systems.

DATA:	LI and LII - D.Init, D.Deval LII - D.Proc, D.SysDAT, D.SysPrc
SYSTEM CALLS:	LI - F$Link, I$Detach*, F$IOQU*, F$SrqMem LII - F$SLink, F$Link, I$Detach*, F$IOQu*, F$SrqMem, F$ID*

11.3.2. I$ChgDir - Change working directory

ASSEMBLER CALL:	OS9 I$ChgDir
MACHINE CODE:	103F 86
ROUTINE LOCATION:	LI and LII - IOMAN
INPUT:	(A) = Access mode.

(X) = Address of the pathlist.

OUTPUT: (X) = Updated past pathlist

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

POSSIBLE ERRORS: LI and LII - E$BPNam, E$BMode

FUNCTION: ChgDir changes a process' working directory to another directory file specified by the pathlist. Depending on the access mode given, the current execution, the current data directory may be changed, or both. The file specified must be a directory file, and the caller must have access permission for the specified mode.

ACCESS MODES:

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

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

Note: The shell “CHD” directive uses UPDATE mode, which means you must have both read and write permission to change directories from the shell.

DATA: LI and LII - D.PrhDBT, D.Proc
LII - D.SysPrc

SYSTEM CALLS: LI - F$All64, F$PrsNam, I$Attach, F$Ret64, F$IOQu, F$Send*, F$Find64*, I$Detach*
LII - F$All64, F$LDABX*, F$PrsNam, I$Attach, F$Ret64, F$IOQU, F$Send, F$GProcP

11.3.3. 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.

POSSIBLE ERRORS: LI and LII - E$BPNum

FUNCTION: Close terminates the I/O path specified by the path number. The path number will no longer be valid for any OS-9 calls unless it unless it becomes active again via I$Open, I$Create, or I$Dup. Devices that are non-sharable become available to other requesting processes. All OS-9 internally managed buffers and descriptors are deallocated.

Note: Because the OS9 F$Exit service request automatically closes all open paths, it may not be necessary to close them individually with the OS9 I$Close service request.

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

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI and LII - F$Find64, I$Detach*, F$Ret64*, F$IOQu, F$Send*
LII - F$GProcP*

CAVEATS: I$Close does an implied I$Detach call. If it causes the device use count to become zero, the device termination routine will be executed. See I$Detach.

11.3.4. I$Create - Create a path to a new file

ASSEMBLER CALL: OS9 I$CREATE

MACHINE CODE: 103F 83

INPUT: (A) = Access mode.
(B) = File attributes (access permission).
(X) = Address of the pathlist.

OUTPUT: (A) = Path number.
(X) = Updated past the pathlist (trailing blanks skipped)

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

POSSIBLE ERRORS: LI and LII - E$PthFul, E$BPNam

FUNCTION: CREATE is used to create a new file on a multifile mass storage device. On non-multifile devices, Create is synonymous with Open. 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 = nonsharable file

The access mode parameter passed in register A must have the write bit set if any data is to be written to the file. This only affects the file until it is closed; it can be reopened later in any access mode allowed by the fil attributes (see OPEN). 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, directory searching will begin with the working execution directory instead of the working data directory.

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

Data storage is allocated for the file automatically by WRITE or explicitly by the PUTSTAT call.

An error will occur if the pathlist specifies a file name that already exists in. Create cannot be used to make directory files (see I$MakDir).

DATA: LI and LII - D.Proc, D.PthDBT

LII - D.SysPrc

SYSTEM CALLS: LI - F$All64, F$PrsNam, I$Attach, F$Ret64*, F$IOQu*, F$Send*, F$Find64*
LII - F$All64, F$LDABX*, F$PrsNam, I$Attach, F$Ret64, F$IOQu*, F$Send*, F$GProcp*

CAVEATS: Create causes an implicit I$Attach call. If the device has not previously been attached the device's initialization routine will be called.

11.3.5. I$Delete - Delete a file

ASSEMBLER CALL: OS9 I$DELETE

MACHINE CODE: 103F 87

ROUTINE LOCATION: LI and LII - I/O

INPUT: (X) = Address of pathlist.

OUTPUT: (X) = Updated past pathlist (trailing spaces skipped).

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

POSSIBLE ERRORS: LI and LII - E$BPNam

FUNCTION: This service request deletes the file specified by the pathlist. The caller must have non-sharable write access to the file or an error will result (i.e. The file may not be open by any process). Attempts to delete non-multifile devices will result in an error.

DATA: LI and LII - D.PthDBT, D.Proc
LII - D.SysPrc

SYSTEM CALLS: LI - I$Detach*, F$Ret64*, F$All64, F$PrsNam, I$Attach, F$IOQu, F$Find64, F$Send*
LII - I$Detach*, F$Ret64*, F$All64, F$LDABX*, F$PrsNam, I$Attach, F$GProcP, F$IOQu*, F$Send*

11.3.6. 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.

FUNCTION: This service request deletes the file specified in the pathlist. I$DeletX is identical to I$Delete except that it accepts a mode byte, which allows the caller to specify the execution directory. Not being able to specify a mode byte was an oversight in earlier versions of OS-9. I$DeletX is the preferred system call to delete files.

The caller must have non-sharable write access to the file or an error will result. Attempts to delete devices will result in error.

The access mode is used to specify the current working directory or the 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 is given (a pathlist beginning with '/'), the access mode is ignored.

DATA: LI and LII - D.Proc, D.PthDBT
LII ONLY - D.SysPrc

SYSTEM CALLS: LI and LII - F$All64, F$PrsNam*, I$Attach, F$Ret64, F$IOQu*, File Mgr, F$Send*, F$Find64*, I$Detach
LII ONLY - F$LDABX*, F$GProcP*

11.3.7. I$Detach - Remove a device from the system

ASSEMBLER CALL: OS9 I$DETACH

MACHINE CODE: 103F 81

ROUTINE LOCATION: LI and LII - I/O

INPUT: (U) = Address of the device table entry.

OUTPUT: None.

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

FUNCTION: LII - 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.

LI - Performs unlink of all three I/O modules associated with the device (driver, descriptor, file manager). Unlink then calls IODel to complete the device termination.

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. SCF also uses ATTACH/DETACH to setup its second (echo) device.

DATA: LI - None.
LII - D.Init, D.Deval, D.Proc, D.SysPrc, D.SysDAT

SYSTEM CALLS: LI - F$UnLink*
LII - F$RtMem*, F$Send*, F$GProcP*, F$Unlink*

11.3.8. I$Dup - Duplicate a path

ASSEMBLER CALL: OS9 I$DUP

MACHINE CODE: 103F 82

ROUTINE LOCATION: LI-and LII - I/O

INPUT: (A) = Path number of path to duplicate.

OUTPUT: (A) = New path number.

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

POSSIBLE ERRORS: LI and LII - E$PthFul, E$BPNum

FUNCTION: Given the number of an existing path, DUP returns a 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. It is usually not a good idea for more than one process to be doing 1/0 on the same path concurrently. On RBF files, unpredictable results may be produced.

NOTE: This only increments the “use count” of a path descriptor and returns the synonymous path number. The path descriptor is NOT copied. It is usually not a good idea for more than process to be doing 1/0 on the same path concurrently. On RBF files, unpredictable results may be produced.

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI and LII - F$Find64

CAVEATS: The DUP will always use the lowest available path number. For example, if the user does I$Close on path #0, then does I$Dup on path #4, then path #0 will be returned as the new path number. In this way, the standard I/O paths may be manipulated to contain any desired paths. In this way, the standard I/O paths may be manipulated to contain ant desired paths.

11.3.9. I$GetStt - Get file/device status

ASSEMBLER CALL: OS9 I$GetStt

MACHINE CODE: 103F 8D

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Path number.
(B) Function code.
(Other registers depend upon status code)

OUTPUT: (depends upon function code)

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

POSSIBLE ERRORS: LI and LII - E$BPNum

FUNCTION: 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 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. (IOMAN - LI, LII)
SS.Ready	$1	Test for data ready. (RBF, Acia - LI, LII)
SS.Size	$2	Return current file size (RBF - LI, LII)
SS.Pos	$5	Get current file position. (RBF - LI, LII)

MNEMONIC	CODE	FUNCTION
SS.EOF	$6	Test for end of file. (RBF, Acia - LI, LII)
SS.DevNm	$E	Return device name. (IOMAN - LI, LII)
SS.FD	$F	Read file descriptor sector. (RBF - LII)

CODES 10-127 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.

This call is handled mostly by IOMAN and should work with all File Managers.

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.

FUNCTION: 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 SCF supported devices.

RBF devices always return Ready.

INPUT: (A) = Path number.
(B) = Function code 1

OUTPUT:

	Ready	Not Ready	Error
(CC)	C bit clear	C bit set	C bit set
(B)	Zero	$F6 (E$NRDY)	Error code

SS.Size (code $2): Get current file Size (RBF 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 (RBF 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.

SCF never returns EOF

INPUT: (A) = Path number.
(B) = Function code 6

OUTPUT:

	Not EOF	EOF	Error
(CC)	C bit clear	C bit set	C bit set
(B)	Zero	$D3 (E$EOF)	Error code

SS.DevNm (code $E) Return device name

INPUT: (A) = Path number
(B) = Function code $E
(X) = Address of 32 byte area for device name

OUTPUT: Device name in 32 byte storage area

SS.FD (code $F) Read FD sector

INPUT: (A) = Path number
(B) = Function code $F
(X) = Address of 256 byte area for FD.
(Y) = Number of bytes to read (<=256).

OUTPUT: File descriptor placed in reserved area.

11.3.10. I$MakDir - Make a new directory

ASSEMBLER CALL: OS9 I$MAKDIR

MACHINE CODE: 103F 85

ROUTINE LOCATION: LI and LII - I/O

INPUT: (B) = Directory attributes.
(X) = Address of pathlist.

OUTPUT: (X) = Updated past pathlist (trailing spaces skipped).

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

POSSIBLE ERRORS: LI and LII - E$BPNam, E$CEF

FUNCTION: 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 directoryA file contains no entries, except for an entry for itself (".") and its parent directory (".."). MAKDIR will fail on non-multifile devices.

The caller is made the owner of the directory. MAKDIR does not return a path number because directory files are not “opened” by this request (use 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 = nonsharable directory
bit 7 = (don't care)

DATA: LI and LII - D.Proc, D.PthDBT
LII - D.SysPrc

SYSTEM CALLS: LI - F$All64, F$PrsNam*, I$Attach, F$Ret64*, F$IOQu*, F$Send*, F$Find64*, FileMgr., I$Detach*
LII - F$All64, F$LDABX*, F$PrsNam*, I$Attach, F$Ret64*, F$IOQu*, F$Send*, F$GProcP*, FileMgr., I$Detach*

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

ASSEMBLER CALL: OS9 I$OPEN

MACHINE CODE: 103F 84

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Access mode (D S PE PW PR E W R)
(X) = Address of pathlist.

OUTPUT: (A) = Path number.
(X) = Updated past pathlist (trailing spaces skipped).

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

POSSIBLE ERRORS: LI and LII - E$PthFul, E$BPNam

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

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

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

For RBF devices, Read mode should be used if preference to Update if the is not going to be modified. This will inhibit record locking, and could dramatically improve system performance if more than one user is accessing the file. The access mode must conform to the access permission attributes associated with the file or device (see CREATE). Only the owner may access a file unless the appropriate “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.

DATA: LI and LII - D.Proc, D.PthDBT
LII - D.SysPrc

SYSTEM CALLS: LI - F$All64, F$PrsNam, I$Attach, F$Ret64*, F$IOQu*, F$Send*, F$Find64

LII - F$All64, F$LDAEX*, F$PrsNam, I$Attach, F$Ret64*, F$IOQu, F $Send*, F$GProcP

CAVEATS: 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).

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

LII - If the non-sharable bit is set, the file will be opened for non- sharable access regardless if the file is sharable.

Directory files may be opened for read or write if the `D` bit (bit 7) is set in the access mode.

Open will always use the lowest path number available for the process during the open.

11.3.12. I$Read - Read data from a file or device

ASSEMBLER CALL: OS9 I$READ

MACHINE CODE: 103F 89

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Path number.
(X) = Address to store data.
(Y) = Maximum number of bytes to read.

OUTPUT: (Y) Number of bytes actually read.

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

POSSIBLE ERRORS: LI - E$BPNum, E$BMode
LII - E$BPNum, E$Read, E$BMode

FUNCTION: 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. If there is not enough data in the file to satisfy the read request, fewer bytes will be read than requested, but an end of file error is not returned.

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

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI and LII - F$Find64, F$IOQu*, FileMgr, F$Send*
LII - F$GProcP

CAVEATS: The keyboard X-ON, X-OFF characters may be filtered out of the input data on SCF-type devices unless the corresponding entries in the path descriptor have been set to zero. It may be desirable to modify the device descriptor so that these values in the path descriptor are initialized to zero when the path is opened.

For LII RBF devices, if the file is open for Update, the record read will be locked out. See the Record Locking section.

The number of bytes requested will be read unless:

A. An end-of-file occurs

B. An end-of-record occurs (SCF only)

C. An error condition occurs.

11.3.13. I$ReadLn - Read a text line with editing

ASSEMBLER CALL:	OS9 I$READLN
MACHINE CODE:	103F 8B
ROUTINE LOCATION:	LI and LII - I/O
INPUT:	(A) = Path number. (X) = Address to store data. (Y) = Maximum number of bytes to read.
OUTPUT:	(Y) Actual number of bytes read.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	LI - E$BPNum, E$BMode LII - E$BPNum, E$Read, E$BMode

FUNCTION: READLN is similar to “READ” except it reads data from the input file or device until a carriage return character is encountered. Also, ReadLn causes line editing to occur on SCF-type devices. Line editing refers to backspace, line delete, echo, automatic line feed, etc.

SCF requires that the last byte entered be an end-of-record character (normally carriage return). If more data is entered than the maximum specified, it will not be accepted and a PD.OVF character (normally bell) will be echoed. For example, a ReadLn of exactly one byte will accept only a carriage return to return without error.

After all data in a 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.

DATA:	LI and LII - D.Proc, D.PthDBT
SYSTEM CALLS:	LI and LII - F$Find64, I$IOQu*, FileMgr, F$Send* LII - F$GProcP*

11.3.14. I$Seek - Reposition the logical file pointer

ASSEMBLER CALL:	OS9 I$SEEK
MACHINE CODE:	103F 88
ROUTINE LOCATION:	LI and LII - I/O
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.

POSSIBLE ERRORS: LI and LII - E$BPNum

FUNCTION: SEEK repositions the path's “file pointer”; which is the 32- bit address of the the next byte in the file to be read or written.

A seek may be performed to any value even if the file is not large enough. Subsequent WRITEs will automatically expand the file to the required size (if possible), but READS will return an end-of-file condition. Note that a SEEK to address zero is the same as a “rewind” operation.

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

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI and LII - F$Find64, File Mgr, F$IOQu*, F$Send*
LII - F$GProcP*

CAVEATS: On RBF devices, seeking to a new disk sector causes the internal disk buffer to be rewritten to disk if it has been modified. Seek does not change the state of record locking.

11.3.15. I$SetStt - Set file/device status

ASSEMBLER CALL: OS9 I$SETSTT

MACHINE CODE: 103F BE

INPUT: (A) = Path number.
(B) = Function code.
(Other registers depend upon the function code).

OUTPUT: (Depends upon the function code).

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

POSSIBLE ERRORS: LI and LII - E$BPNum

FUNCTION: This system call 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 GETSTT service request which is used to read the device's operating parameters etc. Below are the presently defined function codes:

MNEMONIC	CODE	FUNCTION
SS.OPT	$0	Write the 32 byte option section of the path descriptor (SCF,RBF - LI,LII)
SS.Size	$2	Set the file size (RBF - LI,LII)
SS.Reset	$3	Restore head to track zero*
SS.WTrk	$4	Write (format) track*
SS.Feed	$9	Issue Form Feed (SCF)
SS.FRZ	$A	Freeze DD. information*

MNEMONIC	CODE	FUNCTION
SS.SPT	$B	Set Sectors per track*
SS.SQD	$C	Sequence down disk drive*
SS.DCmd	$D	Direct command to hard disk controller*
SS.FD	$F	Write FD sector (RBF - LII)
SS.Ticks	$10	Set Lockout honor duration (RBF - LII)
SS.Lock	$11	Lock/Release record (RBF - LII)
SS.SSIG	$1A	Send signal on data ready (ACIA - LII)
SS.Relea	$1B	Release device (ACIA - LII)

* These setstats exist in Microware supplied disk drivers (if needed). Only SS.Reset and SS.WTrk are required; the others are implemented to allow reading nonstandard disks. Microware has not supplied any software which makes use of them.

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

SS.OPT (code 0): Write option section of path descriptor.

INPUT: (A) = Path number
(B) = Function code 0
(X) = Address of a 32 byte status packet

OUTPUT: None.

FUNCTION: 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. This call is handled by the File Managers, only copies values that are appropriate to be changed by user programs.

SS.SIZE (code 2): Set file size (RBF-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.

FUNCTION: 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

FUNCTION: 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

(Y) - Side/density

For Floppy: Y = side/density
For Hard Disk Y = side only (may be more than one)

Bit B0 = SIDE (0 = side zero, 1 = side one)
Bit B1 = DENSITY (0 = single, 1 = double)

OUTPUT: None

FUNCTION: 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: (A) = Path number
(B) = SS.FRZ function code

OUTPUT: None

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

SS.SPT (Code $B): Set Sectors Per Track

INPUT: (A) = Path number
(B) = SS.SPT function code
(X) = new sectors per track

OUTPUT: None

FUNCTION: Sets a different number of sectors per track so non-standard disks may be read.

SS.SQD (Code $C): Sequence Down Disk

INPUT: (A) = path number
(B) = SS.SQD function code

OUTPUT: None

FUNCTION: Initiates power-down sequence for Winchester or other hard disks which have sequence-down requirements prior to removal of power.

SS.DCmd (Code $D): Direct Command to Disk Controller

INPUT: Varies

OUTPUT: Varies

FUNCTION: Transmits a command directly to an intelligent disk controller for special functions. Parameters and commands are hardware dependent for specific systems.

SS.FD (Code $F): Write FD sector

INPUT: (A) = Path number
(B) = Function code
(X) = Address of FD sector image

OUTPUT: None

FUNCTION: Change FD sector

NOTE: Only FD.OWN, FD.DAT, and FD.Creat can be changed. These are the only fields written back to disk. If at least 16 bytes are not read using the GETSTT call, garbage could be written out to the FD sector.

SS.Lock (Code $10): Lock out a section of a file.

INPUT:	(A) = path number (B) = SS.Lock code (X) = M.S. of lockout size (U) = L.S. of lockout size
OUTPUT:	None

FUNCTION: SS.Lock locks out a section of the file from the current position up to the number of bytes requested. If 0 bytes are requested, all locks (Record Lock, EOF Lock, and File Lock) are removed. If (X) and (U) contain SFFFF FFFF, then the entire file is locked out regardless of where the file pointer is. This is a special type of file lock that remains in effect until released by SS.Lock(O), a read or write of zero bytes, or the file is closed. There is no way to gain file lock using only Read or Write system calls.

SS.Ticks (Code $11) Wait specified number of ticks for record release.

INPUT:	(A) = path number (B) = SS.Ticks code (X) = Delay interval
OUTPUT:	None.

FUNCTION: Normally, if a read or write request is issued for part of a file that is locked out by another user, RBF sleeps indefinately until the conflict is removed. The SS.Ticks call may be used to cause an error (#252) to be returned to the user program if the conflict still exists after the specified number of ticks of the system clock have elapsed.

The delay interval is used directly as a parameter to RBF's conflict sleep request. The value zero (RBF's default) causes a sleep forever until the record is released. A delay value of one effectively means that if the lock is not released immediately, an error is returned.

SS.SSIG (Code $1A) Send Signal on data ready

INPUT:	(A) = path number (B) = function code (X) = user defined signal code
OUTPUT:	None

FUNCTION: SS.SSIG sets up a signal to be sent to a process when a device has data ready. SS.SSIG must be reset each time the signal is sent if it is to be used again. The device is considered busy, and will return an error if any read request arrives before the signal is sent. Write requests are allowed to the device while in this state.

SS.Relea (Code $1B) Release device

INPUT:	(A) = path number (B) = function code
OUTPUT:	None

FUNCTION: SS.Relea clears the the signal to be sent from a device so it will no longer send a signal on data ready.

11.3.16. I$Write - Write data to a file or device

ASSEMBLER CALL: OS9 I$WRITE

MACHINE CODE: 103F 8A

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Path number.
(X) = Address of data to write.
(Y) = Number of bytes to write.

OUTPUT: (Y) Number of bytes actually written.

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

POSSIBLE ERRORS: LI - E$BPNum, E$BMode
LII - E$BPNum, E$BMode, E$Write

FUNCTION: 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.

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI - F$Find64, F$IOQu*, File Mgr, F$Send*
LII - F$GProcP*

11.3.17. I$WritLn - Write a line of text with editing

ASSEMBLER CALL: OS9 I$WRITLN

MACHINE CODE: 103F 8C

ROUTINE LOCATION: LI and LII - I/O

INPUT: (A) = Path number.
(X) = Address of data to write.
(Y) = Maximum number of bytes to write.

OUTPUT: (Y) = Actual number of bytes written.

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

POSSIBLE ERRORS: LI and LII - E$BPNum, E$BMode
LII - E$Write

FUNCTION: This system call is similar to 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.

DATA: LI and LII - D.Proc, D.PthDBT

SYSTEM CALLS: LI and LII - F$Find64, F$IOQu*, File Mgr, F$Send*
LII - F$GProcP

Chapter 12. Level Two System Service Requests

12.1. Level Two System Service Requests

12.1.1. F$AllImg - Allocate Image RAM blocks

ASSEMBLER CALL:	OS9 F$AllImg
MACHINE CODE:	103F 3A
ROUTINE LOCATION:	OS9p1
INPUT:	(A) = Beginning block number (B) = Number of blocks (X) = Process Descriptor pointer
OUTPUT:	None.
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	E$MemFul

FUNCTION: ALLIMG allocates RAM blocks for process DAT image. The blocks may not be contiguous. This call is used to allocate data area for a process.

DATA: D.BlkMap

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.2. F$AllPrc - Allocate Image RAM blocks

ASSEMBLER CALL:	OS9 F$AllPrc
MACHINE CODE:	103F 4B
ROUTINE LOCATION:	OS9p2
INPUT:	None
OUTPUT:	(U) = Process Descriptor pointer
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	E$PrcFul

FUNCTION: ALLPROC allocates and initializes a BIZ-byte process descriptor. The address of the descriptor is kept in the process descriptor table. Initialization consists of clearing the first 256 bytes of the descriptor, setting up the state as system state, and marking as unallocated as much of the DAT image as the system allows (typically 60 - 64k).

DATA: D.PrcDBT

SYSTEM CALLS: F$SrqMem

Note

This is a privileged system mode service request.

12.1.3. F$AllRAM - Allocate RAM blocks

ASSEMBLER CALL: OS9 F$AllRAM

MACHINE CODE: 103F 39

ROUTINE LOCATION: OS9p1

INPUT: (B) = Number of blocks

OUTPUT: None.

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

POSSIBLE ERRORS: E$NoRam

FUNCTION: ALLRAM searches the Memory Block map for the desired number of contiguous free RAM blocks.

DATA: D.BlkMap

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.4. F$AllTsk - Allocate process Task number

ASSEMBLER CALL: OS9 F$AllTsk

MACHINE CODE: 103F 3F

ROUTINE LOCATION: OS9p1

INPUT: (X) = Process Descriptor pointer

OUTPUT: None.

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

POSSIBLE ERRORS: E$NoTask

FUNCTION: ALLTSK determines if a task number is assigned to the given process. Otherwise, a task number is newly allocated, and the DAT image is copied into the DAT hardware.

DATA: D.Tasks, D.SysTsk

SYSTEM CALLS: F.ResTsk, F.SetTsk*

Note

This is a privileged system mode service request.

12.1.5. F$Boot - Bootstrap system

ASSEMBLER CALL: OS9 F$Boot

MACHINE CODE: 103F 35

ROUTINE LOCATION: OS9p1

INPUT: None

OUTPUT: None.

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

FUNCTION: BOOT links 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.

DATA: D.Boot, D.Init, D.SysDAT

SYSTEM CALLS: F$Link, Entry point of Boot Module, F$VModul

Note

This is a privileged system mode service request.

12.1.6. F$BtMem - Bootstrap Memory request

ASSEMBLER CALL: OS9 F$BtMem

MACHINE CODE: 103F 36

ROUTINE LOCATION: OS9p1

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.

POSSIBLE ERRORS: E$MemFul

FUNCTION: BTMEM allocates requested memory (rounded up to nearest block) as contiguous memory in the system's address space.

With the release of version 1.2 of Level Two, this system call is not needed. It is equated to F$SrqMem.

Note

This is a privileged system mode service request.

12.1.7. F$ClrBlk - Clear specific Block

ASSEMBLER CALL: OS9 F$ClrBlk

MACHINE CODE: 103F 50

ROUTINE LOCATION: OS9p2

INPUT: (B) = Number of blocks
(U) = Address of first block

OUTPUT: None.

ERROR OUTPUT: None.

POSSIBLE ERRORS: E$IBA if the address is invalid, or if clearing area where the stack is residing.

FUNCTION: CLRBLK marks blocks in process DAT image as unallocated. Thus, the blocks become free for the process to use for other data or program area.

DATA: D.Proc

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.8. F$CpyMem - Copy external Memory

ASSEMBLER CALL: OS9 F$CpyMem

MACHINE CODE: 103F 18

ROUTINE LOCATION: OS9p2

INPUT: (D) = DAT image ptr.
(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.

FUNCTION: CPYMEM reads external memory into the user's buffer for inspection. Any memory in the system may be viewed in this way. CpyMem assumes X to be the address of the 64k space described by the DAT image given. only the part of the DAT image which corresponds to the area specified needs to be set up.

Examples: If the entire DAT image of a process is passed, then X = address in the process space. If a partial DAT image is passed (upper half), then X = offset from beginning of DAT image ($8000).

DATA: D.TmpDAT, D.Procc

SYSTEM CALLS: F$LDDDXY*, F$LDAXY*, F$STABX*

Note

This is a *user mode* service request.

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

ASSEMBLER CALL: OS9 F$DATLog

MACHINE CODE: 103F 44

ROUTINE LOCATION: OS9p1

INPUT: (B) = DAT image offset
(X) = Block offset

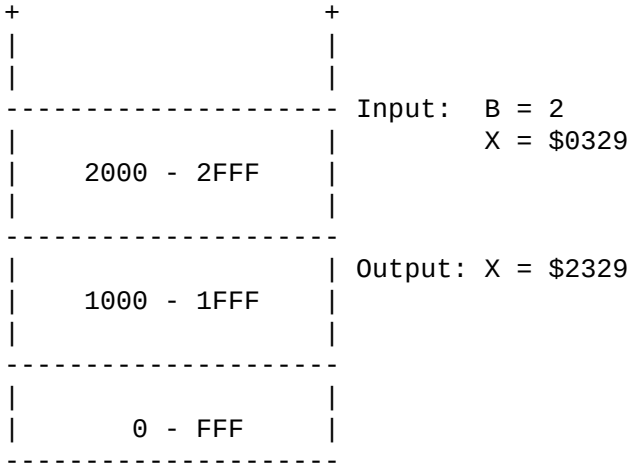
OUTPUT: (X) = Logical address

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

FUNCTION: DATLOG converts a DAT image block number and block offset to its equivalent logical address.

Note

This is a privileged system mode service request.



12.1.10. F$DelImg - Deallocate Image RAM blocks

ASSEMBLER CALL: OS9 F$DelImg

MACHINE CODE: 103F 3B

ROUTINE LOCATION: OS9p2

INPUT: (A) = Beginning block number
(B) = Block count
(X) = Process descriptor pointer

OUTPUT: None.

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

FUNCTION: DELIMG deallocates memory from the process' address space. This call frees the RAM for system use and frees the DAT image for the process. It's main use is to allow the system to clean up after process death.

DATA: D.BlkMap

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.11. F$DelPrc - Deallocate Process descriptor

ASSEMBLER CALL: OS9 F$DelPrc

MACHINE CODE: 103F 4C

ROUTINE LOCATION: OS9p2

INPUT: (A) = Process ID

OUTPUT: None.

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

FUNCTION: DELPRC returns process descriptor memory to a free memory pool. Used for process clean up after death.

DATA: D.PrcDBT

SYSTEM CALLS: F$DelTsk*, F$SrtMem

Note

This is a privileged system mode service request.

12.1.12. F$DelRam - Deallocate RAM blocks

ASSEMBLER CALL: OS9 F$DelRam

MACHINE CODE: 103F 51

ROUTINE LOCATION: OS9p2

INPUT: (B) = Number of blocks
(X) = Starting block number

OUTPUT: None.

ERROR OUTPUT: None.

FUNCTION: DELRAM clears the block's “RAM in use flag” in the system memory block map. DelRam assumes the blocks are not associated with any DAT image.

DATA: D.BlkMap

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.13. F$DelTsk - Deallocate process Task number

ASSEMBLER CALL: OS9 F$DelTsk

MACHINE CODE: 103F 40

ROUTINE LOCATION: OS9p1

INPUT: (X) = Process descriptor pointer

OUTPUT: None.

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

FUNCTION: DELTSK releases the Task number in use by the process whose descriptor pointer is passed.

DATA: D.SysTsk, D.Tasks

SYSTEM CALLS: F.RelTsk

Note

This is a privileged system mode service request.

12.1.14. F$ELink - Link using module directory Entry

ASSEMBLER CALL: OS9 F$ELink

MACHINE CODE: 103F 4D

ROUTINE LOCATION: OS9p1

INPUT: (B) = Module type
(X) = Pointer to module directory entry

OUTPUT: (U) = Module header address
(Y) = Module entry point

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

POSSIBLE ERRORS: E$ModBsy, E$MemFul

FUNCTION: ELINK 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.

DATA: D.Proc

SYSTEM CALLS: F.FreeHB*, F.SetImg*, F.DATLog*, F.LDDDXY*

Note

This is a privileged system mode service request.

12.1.15. F$FModul - Find Module directory entry

ASSEMBLER CALL: OS9 F$FModul

MACHINE CODE: 103F 4B

ROUTINE LOCATION: OS9p1

INPUT: (A) = Module type
(X) = Pointer to name string

	(Y) = DAT image pointer (for name)
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.
POSSIBLE ERRORS:	EMNF, EBNam.

FUNCTION: FMODUL returns a pointer to the module directory entry for the first module whose name and type matches the given name and type. A module type of zero passed to FModul will find any module.

DATA:	D.ModDir, D.ModEnd
SYSTEM CALLS:	F.LDAXY*, F.DATLOg*, F.PrsNam, F.LDDDXY*, F.ChkNam*

Note

This is a privileged system mode service request.

12.1.16. F$FreeHB - Get Free High block

ASSEMBLER CALL:	OS9 F$FreeHB
MACHINE CODE:	103F 3E
ROUTINE LOCATION:	OS9p1
INPUT:	(B) = Block count (Y) = DAT image pointer
OUTPUT:	(A) = Beginning block number
ERROR OUTPUT:	(CC) = C bit set. (B) = Appropriate error code.
POSSIBLE ERRORS:	E$MemFul

FUNCTION: FREEHB searches the DAT image for the highest set of contiguous free blocks of the given size. FreeHB returns the block number of the beginning memory address of the free blocks.

Note

This is a privileged system mode service request.

12.1.17. F$FreeLB - Get Free Low block

ASSEMBLER CALL:	OS9 F$FreeLB
MACHINE CODE:	103F 3D
ROUTINE LOCATION:	OS9p1
INPUT:	(B) = Block count (Y) = DAT image pointer
OUTPUT:	(A) = Beginning block number

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

POSSIBLE ERRORS: E$MemFul

FUNCTION: FREELB searches the DAT image for the lowest set of contiguous free blocks of the given size.

Note

This is a privileged system mode service request.

12.1.18. F$GBlkMp - Get system Block Map copy

ASSEMBLER CALL: OS9 F$GBlkMp

MACHINE CODE: 103F 19

ROUTINE LOCATION: OS9p2

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.

FUNCTION: GBLKMAP copies the system's memory block map into the user's buffer for inspection. F$GBlkMp is used by Mfree to find how much free memory exists.

DATA: D.BlkMap, D.SysTsk, D.Proc

SYSTEM CALLS: F$Move

Note

This is a *user mode* service request.

12.1.19. F$GModDr - Get Module Directory copy

ASSEMBLER CALL: OS9 F$GModDr

MACHINE CODE: 103F 1A

ROUTINE LOCATION: OS9p2

INPUT: (X) = 2048 byte buffer pointer
(Y) = end of copied module dir
(U) = Start address of Module Directory in the system

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

FUNCTION: GMODDR copies the system's module directory into the user's buffer for inspection. F $GModDr is used by Mdir to look at the module directory.

DATA: D.ModDir, D.ModEnd, D.SysTsk, D.Proc

SYSTEM CALLS: F$Move.

Note

This is a *user mode* service request.

12.1.20. F$GPrDsc - Get Process Descriptor copy

ASSEMBLER CALL: OS9 F$GPrDsc

MACHINE CODE: 103F 18

ROUTINE LOCATION: OS9p2

INPUT: (A) = Requested process ID
(X) = 512 byte buffer pointer

OUTPUT: None.

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

FUNCTION: GPRDSC copies a process descriptor into the calling process' buffer for inspection. There is no way to change data in a process- descriptor. F$GPrDsc is used by the Procs utility to gain information about each existing process.

DATA: D.Proc, D.SysTsk

SYSTEM CALLS: F$GProcP

Note

This is a *user mode* service request.

12.1.21. F$GProcP - Get Process Pointer

ASSEMBLER CALL: OS9 F$GProCP

MACHINE CODE: 103F 37

ROUTINE LOCATION: OS9p2

INPUT: (A) = Process ID

OUTPUT: (B) = Pointer to process descriptor

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

POSSIBLE ERRORS: E$BPrcId

FUNCTION: GPROCP translates a process ID number to the address of its process descriptor in the system address space. Process descriptors exist only in the system task address space. Therefore, the address returned must refer to system address space.

DATA: D.PrcDBT

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.22. F$LDABX - Load A from 0,X in task B

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

FUNCTION: One byte is returned from the logical address in (X) in the given task's address space. LDABX 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.

12.1.23. 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.

FUNCTION: LDAXY returns one data byte in the memory block specified by the DAT image in (Y), offset by (X). LDAXY assumes the DAT image pointer is to the actual block desired and that X is only an offset within the DAT block. Example: X must be less than the size of the block or invalid data will be returned. However, no error check is made for the situation.

Note

This is a privileged system mode service request.

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

ASSEMBLER CALL:	OS9 F$LDDDXY
MACHINE CODE:	103F 48
INPUT:	(D) = Offset to the offset within DAT image (X) = Offset within DAT image (Y) = DAT image pointer
OUTPUT:	(D) = bytes addressed by [D+X,Y]

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

FUNCTION: Loads two bytes from the address space described by the DAT image pointed to by (Y). The bytes are loaded from the address D+X in the address space.

Offsets must be set up to be oriented relative to the first block pointed to the DAT image pointer. If the DAT image pointer is to the entire DAT, then D+X should equal the process address for data. If the DAT image is not the entire image (full 64k), then D+X must be adjusted relative to DAT image beginning.

The use of D+X allows keeping a local pointer within a block, yet offsetting into the DAT image to a block number is specified.

SYSTEM CALLS: F.LDAXY*

Note

This is a privileged system mode service request.

12.1.25. 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.

POSSIBLE ERRORS: E$IBA

FUNCTION: MAPBLK maps specified block(s) into unallocated blocks of process space. Blocks are mapped in from the top down. In other words, new blocks are mapped into the highest address available in the address space. See F$ClrBlk for information on “unmapping”.

DATA: D.Proc

SYSTEM CALLS: F$FreeHB, F$SetImg

Note

This is a privileged system mode service request.

12.1.26. F$Move - Move Data (low bound first)

ASSEMBLER CALL: OS9 F$Move

MACHINE CODE: 103F 38

ROUTINE LOCATION: OS9p1

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.

FUNCTION: MOVE moves data bytes from one address space to another, usually from system to user or vice versa.

Note

This is a privileged system mode service request.

12.1.27. F$RelTsk - Release Task number

ASSEMBLER CALL: OS9 F$RelTsk

MACHINE CODE: 103F 43

ROUTINE LOCATION: OS9p1

INPUT: (B) = Task number

OUTPUT: None.

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

FUNCTION: RELTSK releases the specified DAT task number, which makes the task's DAT hardware available for another user.

DATA: D.SysTsk, D.Tasks

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.28. F$ResTsk - Reserve Task number

ASSEMBLER CALL: OS9 F$ResTsk

MACHINE CODE: 103F 42

ROUTINE LOCATION: OS9p1

INPUT: None.

OUTPUT: (B) = Task number

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

FUNCTION: RESTSK finds free DAT task number, reserves it, and returns the task number to the caller. The caller then (generally) assigns the new task number to a process.

DATA: D.SysTsk, D.Tasks

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.29. F$SetImg - Set Process DAT Image

ASSEMBLER CALL: OS9 F$SetImg

MACHINE CODE: 103F 3C

ROUTINE LOCATION: OS9p1

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.

FUNCTION: SETIMG copies DAT image, or a portion of the DAT image, into the process descriptor. At the same time it sets an image change flag in the process descriptor which guarantees that as process returns from the system call, the hardware will be updated to match the new process DAT image.

DATA: None.

SYSTEM CALLS: None.

Note

This is a privileged system mode service request.

12.1.30. F$SetTsk - Set process Task DAT registers

ASSEMBLER CALL: OS9 F$SetTsk

MACHINE CODE: 103F 41

ROUTINE LOCATION: OS9p1

INPUT: (X) = Process descriptor pointer

OUTPUT: None.

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

FUNCTION: SETTSK sets the process task hardware DAT registers and clears the image change flag in the process descriptor as well as writing the data into the hardware.

Note

This is a privileged system mode service request.

12.1.31. F$Slink - System Link

ASSEMBLER CALL: OS9 F$SLink

MACHINE CODE: 103F 34

ROUTINE LOCATION: OS9p1

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.

POSSIBLE ERRORS: E$ModBsy, E$MemFul

FUNCTION: SLINK links a module whose name is outside the current (system) process' address space into the address space. F$SLink is used by the I/O system to get the modules specified by the device name in a user call (I$Create, I$Open, etc.) linked into the system's address space.

DATA: D.Proc

SYSTEM CALLS: F.FModul, F.FreeHB*, F.SetImg*, F.DATLog*, F.LDDXY*

Note

This is a privileged system mode service request.

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

ASSEMBLER CALL: OS9 F$STABX

MACHINE CODE: 103F 4A

ROUTINE LOCATION: OS9p1

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.

FUNCTION: 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.

12.1.33. F$SUser - Set User ID number

ASSEMBLER CALL: OS9 F$SUser

MACHINE CODE: 103F 1C

ROUTINE LOCATION: OS9p1

INPUT: (Y) = Desired user ID number

OUTPUT: None.

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

FUNCTION: SUSER alters the current user ID to that specified, without error checking and without regard to ID number of caller.

DATA: D.Proc

SYSTEM CALLS: None.

Note

This is a *user mode* service request.

12.1.34. F$UnLoad - Unlink module by name

ASSEMBLER CALL: OS9 F$UnLoad

MACHINE CODE: 103F 1D

ROUTINE LOCATION: OS9p2

INPUT: (A) = Module type
(X) = Module name pointer

OUTPUT: None.

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

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

DATA: D.Proc, D.SysDAT, D.BlkMap, D.ModDir, D.ModEnd

SYSTEM CALLS: F$FModul, F$IODel

Note

This is a *user mode* service request.

Appendix A. Standard Floppy Disk Formats

Table A.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)	40	FF	40	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)	85	FF	380	FF
BYTES/SECTOR (FORMATTED)	256		256	
BYTES/TRACK (FORMATTED)	2560		4096	
BYTES/DISK (FORMATTED)	89,600		315,392	

Table A.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 B. Error Codes

B.1. 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	
$C8	200	E$PthFul PATH TABLE FULL - The file cannot be opened because the system path table is currently full.
$C9	201	E$BPNum ILLEGAL PATH NUMBER - Number too large or for non-existant path.
$CA	202	E$Poll INTERRUPT POLLING TABLE FULL
$CB	203	E$BMode ILLEGAL MODE - attempt to perform I/O function of which the device or file is incapable.
$CC	204	E$DevOvf DEVICE TABLE FULL - Can't add another device
$CD	205	E$BMID ILLEGAL MODULE HEADER - module not loaded because its sync code, header parity, or CRC is incorrect.
$CE	206	E$DirFul MODULE DIRECTORY FULL - Can't add another module
$CF	207	E$MemFul MEMORY FULL - Level One: not enough contiquous RAM free. Level Two: process address space full
$D0	208	E$UnkSvc ILLEGAL SERVICE REQUEST - System call had an illegal code number
$D1	209	E$ModBsy MODULE BUSY - non-sharable module is in use by another process.
$D2	210	E$BPAddr BOUNDARY ERROR - Memory allocation or deallocation request not on a page boundary.
$D3	211	E$EOF END OF FILE - End of file encountered on read.
$D4	212	NOT YOUR MEMORY - attempted to deallocate memory not previously assigned.
$D5	213	E$NES NON-EXISTING SEGMENT - device has damaged file structure.
$D6	214	E$FNA FILE NOT ACCESSIBLE - file attributes do not permit access requested.
$D7	215	E$BPNam BAD PATH NAME - syntax error in pathlist (illegal character, etc.).
$D8	216	E$PNNF PATH NAME NOT FOUND - can't find pathlist specified.
$D9	217	E$SLF SEGMENT LIST FULL - file is too fragmented to be expanded further.
$DA	218	E$CEF FILE ALREADY EXISTS - file name already appears in current directory.
$DB	219	E$IBA ILLEGAL BLOCK ADDRESS - device's file structure has been damaged.
$DC	220	ILLEGAL BLOCK SIZE - device's file structure has been damaged.
$DD	221	E$MNF 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.
$DF	223	E$DelSP SUICIDE ATTEMPT - request to return memory where your stack is located.
$E0	224	E$IPrcID ILLEGAL PROCESS NUMBER - no such process exists.
$E2	226	E$NoChld NO CHILDREN - can't wait because process has no children.
$E3	227	E$ISWI ILLEGAL SWI CODE - must be 1 to 3.
$E4	228	E$PrcAbt KEYBOARD ABORT - process aborted by signal code 2.
$E5	229	E$PrcFul PROCESS TABLE FULL - can't fork now.
$E6	230	E$IForkP ILLEGAL PARAMETER AREA - high and low bounds passed in fork call are incorrect.
$E7	231	E$KwnMod KNOWN MODULE - for internal use only.
$E8	232	E$BMCRC INCORRECT MODULE CRC - module has bad CRC value.
$E9	233	E$USigP SIGNAL ERROR - receiving process has previous unprocessed signal pending.
$EA	234	E$NEMod NON-EXISTENT MODULE - unable to locate module.
$EB	235	E$BNam BAD NAME - illegal name syntax.
$EC	236	E$BMHP BAD HEADER - module header parity incorrect
$ED	237	E$NoRam RAM FULL - no free system RAM available at this time
$EE	238	E$BPrcId BAD PROCESS ID - incorrect process ID number
$EF	239	E$NoTask NO TASK NUMBER AVAILABLE - all task numbers in use

B.2. 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	
$F0	240	E$Unit UNIT ERROR - device unit does not exist
$F1	241	E$Sect SECTOR ERROR - sector number is out of range.
$F2	242	E$WP WRITE PROTECT - device is write protected.
$F3	243	E$CRC CRC ERROR - CRC error on read or write verify
$F4	244	E$Read READ ERROR - Data transfer error during disk read operation, or SCF (terminal) input buffer overrun.
$F5	245	E$Write WRITE ERROR - hardware error during disk write operation.
$F6	246	E$NotRdy NOT READY - device has "not ready" status.
$F7	247	E$Seek SEEK ERROR - physical seek to non-existant sector.
$F8	248	E$Full MEDIA FULL - insufficient free space on media.
$F9	249	E$BType WRONG TYPE - attempt to read incompatible media (i.e. attempt to read double-side disk on single-side drive)
$FA	250	E$DevBsy DEVICE BUSY - non-sharable device is in use
$FB	251	E$DIDC DISK ID CHANGE - disk removed and replaced

HEX	DEC	
$FC	252	E$Lock RECORD IS BUSY - record is locked out by another user.
$FD	253	E$Share NON-SHARABLE FILE BUSY - entire file is locked out by another user.
$FE	254	E$DEADLK I/O DEADLOCK ERROR - two processes are attempting to use the same two disk areas simultaneously.

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$Attach	Attach a new device to the system.	
103F 81	I$Detach	Remove a device from the system.	

Code	Mnemonic	Function	Page
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	Fortran I-code

Table C.8. Module Attributes

$8	Reentrant

Appendix D. Direct Page variables

Table D.1. OS-9 Level I Direct Page variables

Name	Offset	Size	Description
D.FMBM	$20	4	Free memory bit map pointers
D.MLIM	$24	2	Memory limit
D.ModDir	$26	4	Module directory
D.Init	$2A	2	Rom base address
D.SWI3	$2C	2	Swi3 vector
D.SWI2	$2E	2	Swi2 vector
D.FIRQ	$30	2	Firq vector
D.IRQ	$32	2	Irq vector
D.SWI	$34	2	Swi vector
D.NMI	$36	2	Nmi vector
D.SvcIRQ	$38	2	Interrupt service entry
D.Poll	$3A	2	Interrupt polling routine
D.UsrIRQ	$3C	2	User irq routine
D.SysIRQ	$3E	2	System irq routine
D.UsrSvc	$40	2	User service request routine
D.SysSvc	$42	2	System service request routine
D.UsrDis	$44	2	User service request dispatch table
D.SysDis	$46	2	System service reuest dispatch table
D.Slice	$48	1	Process time slice count
D.PrcDBT	$49	2	Process descriptor block address
D.Proc	$4B	2	Process descriptor address
D.AProcQ	$4D	2	Active process queue
D.WProcQ	$4F	2	Waiting process queue
D.SProcQ	$51	2	Sleeping process queue
D.Time	$53	6	
D.Year	$53	1	
D.Month	$54	1	
D.Day	$55	1	
D.Hour	$56	1	
D.Min	$57	1	
D.Sec	$58	1	
D.Tick	$59	1	
D.TSec	$5A	1	Ticks / second
D.TSlice	$5B	1	Ticks / time-slice
D.IOML	$5C	2	I/O mgr free memory low bound
D.IOMH	$5E	2	I/O mgr free memory hi bound
D.DevTbl	$60	2	Device driver table addr
D.PolTbl	$62	2	Irq polling table addr

Name	Offset	Size	Description
D.PthDBT	$64	2	Path descriptor block table addr
D.BTLO	$66	2	Bootstrap low address
D.BTHI	$68	2	Bootstrap hi address
D.DMAReq	$6A	1	DMA in use flag
D.AltIRQ	$6B	2	Alternate IRQ vector (CC)
D.KbdSta	$6D	2	Keyboard scanner static storage (CC)
D.DskTmr	$6F	2	Disk Motor Timer (CC)
D.CBStrt	$71	16	Reserved for CC warmstart ($71)
D.Clock	$81	2	Address of Clock Tick Routine (CC)

Table D.2. OS-9 Level II Direct Page variables

Name	Offset	Size	Description
D.Tasks	$20	2	Task User Table
D.TmpDAT	$22	2	Temporary DAT Image stack
D.Init	$24	2	Initialization Module ptr
D.Poll	$26	2	Interrupt Polling Routine ptr
D.Time	$28	6	System Time
D.Year	$28	1	
D.Month	$29	1	
D.Day	$2A	1	
D.Hour	$2B	1	
D.Min	$2C	1	
D.Sec	$2D	1	
D.Tick	$2E	1	
D.Slice	$2F	1	current slice remaining
D.TSlice	$30	1	Ticks per Slice
D.Boot	$31	1	Bootstrap attempted flag
D.MotOn	$32	1	Floppy Disk Motor-On time out
D.ErrCod	$33	1	Reset Error Code
D.Daywk	$34	1	day of week, com-trol clock
D.TkCnt	$35	1	Tick Counter
D.BtPtr	$36	2	Address of Boot in System Address space
D.BtSz	$38	2	Size of Boot
D.BlkMap	$40	4	Memory Block Map ptr
D.ModDir	$44	4	Module Directory ptrs
D.PrcDBT	$48	2	Process Descriptor Block Table ptr
D.SysPrc	$4A	2	System Process Descriptor ptr
D.SysDAT	$4C	2	System DAT Image ptr
D.SysMem	$4E	2	System Memory Map ptr
D.Proc	$50	2	Current Process ptr
D.AProcQ	$52	2	Active Process Queue
D.WProcQ	$54	2	Waiting Process Queue

Name	Offset	Size	Description
D.SProcQ	$56	2	Sleeping Process Queue
D.ModEnd	$58	2	Module Directory end ptr
D.ModDAT	$5A	2	Module Dir DAT image end ptr
D.CldRes	$5C	2	Cold Restart vector
D.Crash	$6B	6	Pointer to CC Crash Routine
D.CBStrt	$71	$B	Reserved for CC warmstart ($71)
D.DevTbl	$80	2	I/O Device Table
D.PolTbl	$82	2	I/O Polling Table
	$84	4	reserved
D.PthDBT	$88	2	Path Descriptor Block Table ptr
D.DMAReq	$8A	1	DMA Request flag
This area is used for the CoCo Hardware Registers			
D.HINIT	$90	1	GIME INIT0 register (hardware setup $FF90)
D.TINIT	$91	1	GIME INIT1 register (timer/task register $FF91)
D.IRQER	$92	1	Interrupt enable register ($FF92)
D.FRQER	$93	1	Fast Interrupt enable register ($FF93)
D.TIMMS	$94	1	Timer most significant nibble ($FF94)
D.TIMLS	$95	1	Timer least significant byte ($FF95)
D.RESV1	$96	1	two reserved registers ($FF96)
D.RESV2	$97	1	($FF97)
D.VIDMD	$98	1	video mode register ($FF98)
D.VIDRS	$99	1	video resolution register ($FF99)
D.BORDR	$9A	1	border register ($FF9A)
D.RESV3	$9B	1	reserved register ($FF9B)
D.VOFF2	$9C	1	vertical scroll/offset 2 register ($FF9C)
D.VOFF1	$9D	1	vertical offset 1 register ($FF9D)
D.VOFF0	$9E	1	vertical offset 0 register ($FF9E)
D.HOFF0	$9F	1	horizontal offset 0 register ($FF9F)
D.Speed	$A0	1	Speed of COCO CPU 0=slow,1=fast
D.TskIPt	$A1	2	Task image Pointer table (CC)
D.MemSz	$A3	1	128/512K memory flag (CC)
D.SSTskN	$A4	1	System State Task Number (COCO)
D.CCMem	$A5	2	Pointer to beginning of CC Memory
D.CCStk	$A7	2	Pointer to top of CC Memory
D.Flip0	$A9	2	Change to Task 0
D.Flip1	$AB	2	Change to reserved Task 1
D.VIRQ	$AD	2	VIRQ Polling routine
D.IRQS	$AF	1	IRQ shadow register (CC Temporary)
D.CLTb	$B0	2	VIRQ Table address
D.AltIRQ	$B2	2	Alternate IRQ Vector (CC)
D.SysSvc	$C0	2	System Service Routine entry
D.SysDis	$C2	2	System Service Dispatch Table ptr

Name	Offset	Size	Description
D.SysIRQ	$C4	2	System IRQ Routine entry
D.UsrSvc	$C6	2	User Service Routine entry
D.UsrDis	$C8	2	User Service Dispatch Table ptr
D.UsrIRQ	$CA	2	User IRQ Routine entry
D.SysStk	$CC	2	System stack
D.SvcIRQ	$CE	2	In-System IRQ service
D.SysTsk	$CF	1	System Task number
D.Clock	$E0	2	
D.XSWI3	$E2	2	
D.XSWI2	$E4	2	
D.XFIRQ	$E6	2	
D.XIRQ	$E8	2	
D.XSWI	$EA	2	
D.XNMI	$EC	2	
D.ErrRst	$EE	2	
D.SWI3	$F2	2	
D.SWI2	$F4	2	
D.FIRQ	$F6	2	
D.IRQ	$F8	2	
D.SWI	$FA	2	
D.NMI	$FC	2	

Colophon

This manual was discovered as a scanned PDF file on the Internet in 2017. The scan was produced by Marcus Bennett in 2008. OCR was then applied to the pages and it was reformatted into Docbook 5.0.

To explain the “DATA” section in the service request descriptions in chapters 11 and 12, an appendix D was added describing the direct page (page 0) variables.