NitrOS-9 Technical Reference

The NitrOS-9 Project
http://www.nitros9.org

Revision History:

Revision	Date	Comment
0.1	July 10, 2004	Created

Acknowledgements:
Input and Typesetting: Boisy Pitre.

Table of Contents

Chapter 1. System Organization

The NitrOS-9 Operating System is composed of groups of modules that work together to perform a common task. The following illustration shows the major modules and their position in the five-layer organization of NitrOS-9:

<table>
<tr><td>Layer 5</td><td>D0</td><td>D1</td><td>D2</td><td>Term</td><td>T1</td><td>T2</td><td colspan="3">Pipe</td></tr>
<tr><td>Layer 4</td><td colspan="3">Floppy Driver (rb1773)</td><td colspan="3">Terminal or Serial Driver</td><td colspan="3">Pipe Driver (Piper)</td></tr>
<tr><td>Layer 3</td><td colspan="3">Disk File Manager (RBF)</td><td colspan="3">Character File Manager (SCF)</td><td colspan="3">Pipe File Manager (PIPEMAN)</td></tr>
<tr><td>Layer 2</td><td colspan="9">Input/Output Manager (IOMan)</td></tr>
<tr><td>Layer 1</td><td colspan="6">NitrOS-9 Kernel (Krn, KrnP2)</td><td>Init</td><td>Clock</td><td>Clock2</td></tr>
</table>

Layer 1: The Kernel

At the lowest layer, **Krn** and **KrnP2** make up the two primary parts of the *kernel*, or core of NitrOS-9. It is the kernel that provides the intelligence behind NitrOS-9, and handles basic system services such as multitasking and memory management. The kernel also links all other NitrOS-9 modules into the system.

Another important set of modules that reside at this layer are **Clock** and **Clock2**. Together, these two modules work to keep track of both system time (known as the *tick*, the heartbeat of the system) as well as actual clock time, either through software or via real-time clock hardware.

The final module of this layer is **Init**. This module contains a table of initialization values and is consulted by the kernel during system startup. Information such as

the user task to run after boot, initial table sizes, and device names are found in this module. It is loaded into RAM (random access memory) by the NitrOS-9 bootstrap module **Boot**, along with other necessary system modules.

Layer 2: IOMan

The system's second layer (just above kernel) contains the input/output manager, **IOMan**. This module provides common processing for all input/output operations, and is required for performing any I/O supported by NitrOS-9.

Layer 3: File Managers

The system's third layer contains *file managers.* File managers perform I/O request processing for similar classes of I/O devices. There are three file managers:

RBF The random block file manager processes all disk I/O operations.

SCF The sequential character file manager handles all non-disk I/O operations that operate one character at a time. These operations include terminal and printer I/O.

PIPEMAN The pipe file manager handles *pipes*. Pipes are memory buffers that act as files. Pipes are used for data transfers between processes.

Layer 4: Device Drivers

The system's fourth layer is the *device driver* layer. Device drivers handle basic I/O functions for specific I/O controller hardware, and are normally provided to you when you purchase new I/O devices or cartridges. You can use pre-written drivers, or you can write your own.

Layer 5: Device Descriptors

The system's fifth layer contains the *device descriptors*. Device descriptors are small tables that define the logical name, device driver and file manager for each I/O port. They also contain port initialization and port address information. Device descriptors require only one copy of each I/O controller driver used.

Beyond Layer 5: Applications

NitrOS-9's primary purpose is to act as the manager of data flow for applications, which run as processes outside of the five layer hierarchy. This includes the initial user process, **SysGo**, which is forked after boot, and **Shell**, the program that allows commands to be typed and executed by NitrOS-9.

Chapter 2. The Kernel

The kernel, as stated in the previous chapter, is the true *core* of NitrOS-9. All resource management and services, from memory allocation to the creation and destruction of processes, are supervised by this very important software component.

The kernel is actually split into two parts: **Krn** (which holds core system calls that must be present during the boot process) and **KrnP2** (which handles additional system calls). These two modules complete the concept of the NitrOS-9 kernel.

The kernel modules for NitrOS-9 Level 1 are smaller than those of NitrOS-9 Level 2, and are small enough to reside on the boot track. Under NitrOS-9 Level 2, **Krn** resides in the boot track while **KrnP2** is part of the OS9Boot file, a file that is loaded into RAM with the other NitrOS-9 modules at bootstrap time.

Here's a look at the kernel's main responsibilities:

- System initialization after reset
- Service request processing
- Memory management
- Multiprogramming management
- Interrupt processing

I/O functions are not included in the list because the kernel does not directly process them. Instead, it passes I/O system calls to the I/O Manager, **IOMan**, for processing.

We will now explore the kernel's responsibilities in more detail.

System Initialization

After a hardware reset, the kernel initializes the system. This involves:

1. Locating modules loaded into memory from the NitrOS-9 boot file.

2. Determining the amount of available RAM.

3. Loading any required modules that were not loaded from the NitrOS-9 boot file.

NitrOS-9 also adds the ability to install new system calls through the **`F$SSvc`** system service call. Under NitrOS-9 Level 1, user state programs can directly call this system call. However, NitrOS-9 Level 2 user processes cannot call this system call directly because it is *privileged*. Instead, new system calls are added through special kernel extension modules, named **KrnP3**, **KrnP4**, **KrnP5**, etc. These kernel modules must be present in the OS9Boot file. The cold start routine in **KrnP2** performs a link to **KrnP3**, and if it exists in the boot file, it will be branched to. If **KrnP3** does not exist in the boot file, **KrnP2** continues with a normal cold start.

System Call Processing

System Calls are used to communicate between NitrOS-9 and programs for such functions as memory allocation and process creation. In addition to I/O and memory management functions, system calls have other functions. These include inter-process control and timekeeping.

System calls use the 6809 microprocessor's SWI2 instruction followed by a constant byte representing the code. You usually pass parameters for system calls in the 6809 registers.

OS9Defs and Symbolic Names

A system-wide assembly language *equate file*, called OS9Defs, defines symbolic names for all system calls. This file is normally included when assembling hand-written or compiler-generated code. The NitrOS-9 assembler has a built-in *macro* to generate system calls. For example:

```
os9 I$Read
```

is recognized and assembled as equivalent to:

```
swi2
fcb  I$Read
```

The NitrOS-9 assembler macro “os9” generates an SWI2 instruction. The label **`I$Read`** is the label for the system call code $89.

Types of System Calls

System calls are divided into two categories: *I/O calls* and *function calls.*

I/O calls perform various input/output functions. The kernel passes calls of this type to the I/O manager for processing. The symbolic names for I/O calls begin with I$ instead of F$. For example, the Read system call is called **`I$Read`**.

Function calls perform memory management, multi-programming and other functions, with most being processed by the kernel. The symbolic names for function calls begin with F$. For example, the Link function call is called **`F$Link`**.

The function calls include *user calls* and *privileged system mode calls.* (See Chapter 8, “System Calls,” for more information.)

Memory Management

Memory management is an important operating system function. Using memory and modules, NitrOS-9 manages the logical contents of memory and the physical assignment of memory to programs.

An important concept in memory management is the *memory module*. The memory module is a format in which programs must reside. NitrOS-9 maintains a *module directory* that points to the modules that occupy memory. This module directory contains information about each module, including its name and address and the number of processes using it. The number of processes using a module is reflected in the module’s *link count*.

When a module’s link count reaches zero, NitrOS-9 releases the module, returns the memory it held back to the free pool, and removes its name from the module directory.

Memory modules are the foundation of NitrOS-9's modular software environment, and have several advantages:

- Automatic runtime linking of programs to libraries of utility modules
- Automatic sharing of re-entrant programs
- Replacement of small sections of large programs into memory for update or correction.

Memory Use in NitrOS-9

NitrOS-9 automatically allocates memory when any of the following occurs:

- Program modules are loaded into RAM
- Processes are created
- Processes execute system calls to request additional RAM
- NitrOS-9 needs I/O buffers or larger tables

NitrOS-9 also has inverse functions to deallocate memory allocated to program modules, new processes, buffers, and tables.

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

NitrOS-9 Level 1 Memory Specifics

Under NitrOS-9 Level 1, a maximum of 64K of RAM is supported. The operating system and all processes must share this memory. In the 64K address map, NitrOS-9 reserves some space at the top and bottom of RAM for its own use. The amount depends on the sizes of system tables that are specified in the **Init** module.

NitrOS-9 pools all other RAM into a free memory space. As the system allocates or deallocates memory, it dynamically takes it from or returns it to this pool. Under NitrOS-9 Level 2, RAM does not need to be contiguous because the memory management unit can dynamically rearrange memory addresses.

The basic unit of memory allocation is the 256-byte page. NitrOS-9 Level 1 always allocates memory in whole numbers of pages.

The data structure that NitrOS-9 uses to keep track of memory allocation is a 256-byte bitmap. Each bit in this table is associated with a specific page of memory. A cleared bit indicates that the page is free and available for assignment. A set bit indicates that the page is in use (that no RAM is free at that address).

NitrOS-9 Level 2 Memory Specifics

Because NitrOS-9 Level 2 utilizes the Memory Management Unit (MMU) component of the Color Computer 3, up to 2MB of memory can be supported. However, each process is still limited to a maximum of 64K of RAM.

Even with this limitation, there is a significant advantage over NitrOS-9 Level 1. Every process has its own 64K "playground." Even the operating system itself has its own 64K area. This means that programs do not have to share a single 64K block with each other or the system. Consequently, larger programs are possible under NitrOS-9 Level 2.

These 64K areas are made up of 8K blocks, the size that is imposed by the MMU found in the Color Computer 3. NitrOS-9 Level 2 assembles a number of these 8K blocks to provide every process (including the system) its own 64K working area.

Within the system's 64K address map, memory is still allocated in 256-byte pages, just like NitrOS-9 Level 1.

Color Computer 3 Memory Management Hardware

As mentioned previously, the 8-bit CPU in the Color Computer 3 can directly address only 64K of memory. This limitation is imposed by the 6809, which has only 16 address lines (A0-A15). The Color Computer 3's Memory Management Unit (MMU) extends the addressing capability of the computer by increasing the address lines to 19 (A0-A18). This lets the computer address up to 512K of memory ($0-$7FFFF), or up to 2MB of memory ($0-$1FFFFF) when enhanced with certain memory upgrades. In this document we will discuss the more common 512K configuration.

The 512K address space is called the *physical address space*. The physical address space is subdivided into 8K *blocks*. The six high order address bits (A13-A18) define a *block number*.

NitrOS-9 creates a *logical address space* of up to 64K for each task by using the **F$Fork** system call. Even though the memory within a logical address space appears to be contiguous, it might not be—the MMU translates the physical addresses to access available memory. Address spaces can also contain blocks of memory that are common to more than one map.

The MMU consists of a multiplexer and a 16 by 6-bit RAM array. Each of the 6-bit elements in this array is an MMU task register. The computer uses these task registers to determine the proper 8-kilobyte memory segment to address.

The MMU task registers are loaded with addressing data by the CPU. This data indicates the actual location of each 8-kilobyte segment of the current system memory. The task register are divided into two sets consisting of eight registers each. Whether the task register select bit (TR bit) is set or reset determines which of the two sets is to be used.

The relation between the data in the task register and the generated addresses is as follows:

Bit	D5	D4	D3	D2	D1	D0
Corresponding Memory Address	A18	A17	A16	A15	A14	A13

When the CPU accesses any memory outside the I/O and control range (XFF00-XFFFF), the CPU address lines (A13-A15) and the TR bit determine what segment of memory to address. This is done through the multiplexer when SELECT is low (See the following table.)

When the CPU writes data to the MMU, A0-A3 determine the location of the MMU register to receive the incoming data when SELECT is high. The following diagram illustrates the operation of the Color Computer 3's memory management.

The system uses the data from the MMU registers to determine the block of memory to be accessed, according to the following table:

TR Bit	A15	A14	A13	Address Range	MMU Address
0	0	0	0	X0000-X1FFF	FFA0
0	0	0	1	X2000-X3FFF	FFA1

0	0	1	0	X4000-X5FFF	FFA2
0	0	1	1	X6000-X7FFF	FFA3
0	1	0	0	X8000-X9FFF	FFA4
0	1	0	1	XA000-XBFFF	FFA5
0	1	1	0	XC000-XDFFF	FFA6
0	1	1	1	XE000-XFFFF	FFA7
1	0	0	0	X0000-X1FFF	FFA8
1	0	0	1	X2000-X3FFF	FFA9
1	0	1	0	X4000-X5FFF	FFAA
1	0	1	1	X6000-X7FFF	FFAB
1	1	0	0	X8000-X9FFF	FFAC
1	1	0	1	XA000-XBFFF	FFAD
1	1	1	0	XC000-XDFFF	FFAE
1	1	1	1	XE000-XFFFF	FFAF

The translation of physical addresses to 8K blocks is as follows:

Range		Block	Range		Block
From	**To**	**Number**	**From**	**To**	**Number**
00000	01FFF	00	40000	41FFF	20
02000	03FFF	01	42000	43FFF	21
04000	05FFF	02	44000	45FFF	22
06000	07FFF	03	46000	47FFF	23
08000	09FFF	04	48000	49FFF	24
0A000	0BFFF	05	4A000	4BFFF	25
0C000	0DFFF	06	4C000	4DFFF	26
0E000	0FFFF	07	4E000	4FFFF	27
10000	11FFF	08	50000	51FFF	28
12000	13FFF	09	52000	53FFF	29
14000	15FFF	0A	54000	55FFF	2A
16000	17FFF	0B	56000	57FFF	2B
18000	19FFF	0C	58000	59FFF	2C
1A000	1BFFF	0D	5A000	5BFFF	2D
1C000	1DFFF	0E	5C000	5DFFF	2E
1E000	1FFFF	0F	5E000	5FFFF	2F
20000	21FFF	10	60000	61FFF	30
22000	23FFF	11	62000	63FFF	31
24000	25FFF	12	64000	65FFF	32
26000	27FFF	13	66000	67FFF	33

28000	29FFF	14	68000	69FFF	34
2A000	2BFFF	15	6A000	6BFFF	35
2C000	2DFFF	16	6C000	6DFFF	36
2E000	2FFFF	17	6E000	6FFFF	37
30000	31FFF	18	70000	71FFF	38
32000	33FFF	19	72000	73FFF	39
34000	35FFF	1A	74000	75FFF	3A
36000	37FFF	1B	76000	77FFF	3B
38000	39FFF	1C	78000	79FFF	3C
3A000	3BFFF	1D	7A000	7BFFF	3D
3C000	3DFFF	1E	7C000	7DFFF	3E
3E000	3FFFF	1F	7E000	7FFFF	3F

In order for the MMU to function, the TR bit at $FF90 must be cleared and the MMU must be enabled. However, before doing this, the address data for each memory segment must be loaded into the designated set of task registers. For example, to select a standard 64K map in the top range of the Color Computer 3's 512K RAM, with the TR bit set to 0, the following values must be preloaded into the MMU's registers:

MMU Location Address	Data (Hex)	Data (Binary)	Address Range
FFA0	38	111000	70000-71FFF
FFA1	39	111001	72000-73FFF
FFA2	3A	111010	74000-75FFF
FFA3	3B	111011	76000-77FFF
FFA4	3C	111100	78000-79FFF
FFA5	3D	111101	7A000-7BFFF
FFA6	3E	111110	7C000-7DFFF
FFA6	3F	111111	7E000-7FFFF

Although this table shows MMU data in the range $38 to $3F, any data between $0 and $3F can be loaded into the MMU registers to select memory addresses in the range 0 to $7FFFF.

Normally, the blocks containing I/O devices are kept in the system map, but not in the user maps. This is appropriate for timesharing applications, but not for process control. To directly access I/O devices, use the **`F$MapBlk`** system call. This call takes a starting block number and block count, and maps them into

unallocated spaces of the process' address space. The system call returns the logical address at which the blocks were inserted.

For example, suppose a display screen in your system is allocated at extended addresses $7A000-$7DFFF (blocks $3D and $3E). The following system call maps them into your address space:

```
ldb     #$02        number of blocks
ldx     #$3D        starting block number
os9     F$MapBlk    call MapBlk
stu     IOPorts     save address where mapped
```

On return, the U register contains the starting address at which the blocks were switched. For example, suppose that the call returned $4000. To access extended address $7A020, write to $4020.

Other system calls that copy data to or from one task's map to another are available, such as **F$STABX** and **F$Move**. Some of these calls are system mode privileged. You can unprotect them by changing the appropriate bit in the corresponding entry of the system service request table and them making a new system boot with the patched table.

Multiprogramming

NitrOS-9 is a multiprogramming operating system. This means that several independent programs called *processes* can be executed at the same time. By issuing the appropriate system call to NitrOS-9, each process can have access to any system resource.

Multiprogramming functions use a hardware real-time clock. The clock generates interrupts 60 times per second, or one every 16.67 milliseconds. These interrupts are called ticks.

Processes that are not waiting for some event are called *active processes*. NitrOS-9 runs active processes for a specific system-assigned period called a time slice. The number of time slices per minute during which a process is allowed to execute depends on a process' priority relative to all other active processes. Many NitrOS-9 system calls are available to create, terminate and control processes.

Process Creation

A process is created when an existing process executes the **`F$Fork`** system call. This call's main argument is the name of the program module that the new process is to execute first (the *primary module*).

Finding the Module. NitrOS-9 first attempts to find the module in the module directory. If it does not find the module, NitrOS-9 usually attempts to load into a memory a mass-storage file in the execution directory, with the requested module name as a filename.

Assigning a Process Descriptor. Once OS-9 finds the module, it assigns the process a data structure called a *process descriptor.* This is a 64-byte package that contains information about the process, its state (see the following section, "Process States"), memory allocations, priority, queue pointers, and so on. NitrOS-9 automatically initializes and maintains the process descriptor.

Allocate RAM. The next step is to allocate RAM for the process. The primary module's header contains a storage size, which NitrOS-9 uses, unless a larger one was requested at fork time. The memory is allocated from the free memory space and given to that process.

Assign Process ID and User ID. NitrOS-9 assigns the new process a unique number called a *process ID*. Other processes can communicate with the process by referring to its ID in various system calls.

The process also has a *user ID*, which is used to identify all processes and files that belong to a particular user. The user ID is inherited from the parent process.

Process Termination. A process terminates when it executes the **`F$Exit`** system call, or when it receives a *fatal* signal. The termination closes any open paths, deallocates memory used by the process, and unlinks its primary module.

Process States

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

- Active – The process is ready for execution.
- Waiting – The process is suspended until a *child process* terminates or until it receives a signal. A child process is a process that is started by another process known as the *parent process*.
- Sleeping – The process is suspended for a specific period of time or until it receives a signal.

Each state has its own queue, a linked list of *descriptors* of processes in that state. To change a process' state, NitrOS-9 moves its descriptor to another queue.

The Active State. Each active process is given a time slice for execution, according to its priority. The scheduler in the kernel ensures that all active processes, even those of low priority, get some CPU time.

The Wait State. This state is entered when a process executes the **`F$Wait`** system call. The process remains suspended until one of its *child* processes terminates or until it receives a *signal*. (See the "Signals" section later in this chapter.)

The Sleep State. This state is entered when a process executes the **`F$Sleep`** system call, which expects the number of ticks for which the process is to remain in the sleep queue. The process will remain until the specified time has elapsed, or until it receives a wakeup signal.

Execution Scheduling

The NitrOS-9 scheduler uses an algorithm that ensures that all active processes get some amount of execution time.

All active processes are members of the *active process queue*, which is kept sorted by process *age*. Age is the number of process switches that have occurred since the process' last time slice. When a process is moved to the active process queue from another queue, its age is set according to its priority—the higher the priority, the higher the age.

Whenever a new process becomes active, the ages of all other active processes increase by one time slice count. When the executing process' time slice has elapsed, the scheduler selects the next process to be executed (the one with the next highest age, the first one in the queue). At this time, the ages of all other active processes increase by one. Ages never go beyond 255.

A new active process that was terminated while in the system state is an exception. The process is given high priority because it is usually executing critical routines that affect shared system resources.

When there are no active processes, the kernel handles the next interrupt and then executes a CWAI instruction. This procedure decreases interrupt latency time (the time it takes the system to process an interrupt).

Signals

A *signal* is an asynchronous control mechanism used for interprocess communication and control. It behaves like a software interrupt, and can cause a process to suspend a program, execute a specific routine, and then return to the interrupted program.

Signals can be sent from one process to another by the **F$Send** system call. Or, they can be sent from NitrOS-9 service routines to a process.

A signal can convey status information in the form of a 1-byte numeric value. Some *signal codes* (values) are predefined, but you can define most. Those already defined by NitrOS-9 are:

0	Kill (terminates the process, is non-interceptable)
1	Wakeup (wakes up a sleeping process)
2	Keyboard terminate
3	Keyboard interrupt
4	Window change
128-255	User defined

When a signal is sent to a process, the signal is saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. When the process gets its next time slice, the signal is processed.

What happens next depends on whether or not the process has set up a *signal intercept trap* (also known as a signal service routine) by executing the **F$Icpt** system call.

If the process has set up a signal intercept trap, the process resumes execution at the address given in the system call. The signal code passes to this routine. Terminate the routine with an RTI instruction to resume normal execution of the process.

Note: *A wakeup signal activates a sleeping process. It sets a flag but ignores the call to branch to the intercept routine.*

If it has not set up a signal intercept trap, the process is terminated immediately. It is also terminated if the signal code is zero. If the process is in the system mode, NitrOS-9 defers the termination. The process dies upon return to the user state.

A process can have a signal pending (usually because the process has not been assigned a time slice since receiving the signal). If it does, and another process tries to send it another signal, the new signal is terminated, and the **F$Send** system call returns an error. To give the destination process time to process the pending signal, the sender needs to execute an **F$Sleep** system call for a few ticks before trying to send the signal again.

Interrupt Processing

Interrupt processing is another important function of the kernel. OS-9 sends each hardware interrupt to a specific address. This address, in turn, specifies the address of the device service routine to be executed. This is called *vectoring* the interrupt. The address that points to the routine is called the *vector*. It has the same name as the interrupt.

The SWI, SWI2, and SWI3 vectors point to routines that read the corresponding pseudo vector from the process' descriptor and dispatch to it. This is why the **F$SSWI** system call is local to a process; it only changes a pseudo vector in the process descriptor.

Vector	Address
SWI3	$FFF2
SWI2	$FFF4
FIRQ	$FFF6
IRQ	$FFF8
SWI	$FFFA
NMI	$FFFC
RESTART	$FFFE

FIRQ Interrupt. The system uses the FIRQ interrupt. The FIRQ vector is not available to you. The FIRQ vector is reserved for future use. Only one FIRQ generating device can be in the system at a time.

Logical Interrupt Polling System

Because most NitrOS-9 I/O devices use IRQ interrupts, NitrOS-9 includes a sophisticated polling system. The IRQ polling system automatically identifies the source of the interrupt, and then executes its associated user- or system-defined service routine.

IRQ Interrupt. Most NitrOS-9 I/O devices generate IRQ interrupts. The IRQ vector points to the real-time clock and the keyboard scanner routines. These routines, in turn, jump to a special IRQ polling system that determines the source of the interrupt. The polling system is discussed in an upcoming paragraph.

NMI Interrupt. The system uses the NMI interrupt. The NMI vector, which points to the disk driver interrupt service routine, is not available to you.

The Polling Table. The information required for IRQ polling is maintained in a data structure called the *IRQ polling table.* The table has an entry for each device that might generate an IRQ interrupt. The table size is permanent and is defined by an initialization constant in the **Init** module. Each entry in the polling table is given a number from 0 (lowest priority) to 255 (highest priority). In this way, the more important devices (those that have a higher interrupt frequency) can be polled before the less important ones.

Each entry has six variables:

Polling Address	Points to the status register of the device. The register must have a bit or bits that indicate if it is the source of an interrupt.
Flip byte	Selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear.
Mask Byte	Selects one or more interrupt request flag bits within the device status register. The bits identify the active task or device.
Service Routine Address	Points to the interrupt service routine for the device. You supply this address.

Static Storage Address	Points to the permanent storage area required by the device service routine. You supply this address.
Priority	Sets the order in which the devices are polled (a number from 0 to 255).

Polling the Entries. When an IRQ interrupt occurs, NitrOS-9 enters the polling system via the corresponding RAM interrupt vector. It starts polling the devices in order of priority. NitrOS-9 loads the status register address of each entry into Accumulator A, using the device address from the table.

NitrOS-9 performs an exclusive-OR operation using the flip byte, followed by a logical-AND operation using the mask byte. If the result is non-zero, NitrOS-9 assumes that the device is the source of the interrupt.

NitrOS-9 reads the device memory address and service routine address from the table, and performs the interrupt service routine.

Note: *If you are writing your own device driver, terminate the interrupt service routine with an RTS instruction,* ***not*** *an RTI instruction.*

Adding Entries to the Table. You can make entries to the IRQ (interrupt request) polling table by using the **`F$IRQ`** system call. This call is a *privileged system call*, and can only be executed in system mode. NitrOS-9 is in system mode whenever it is running a device driver.

Note: *The code for the interrupt polling system is located in the I/O Manager module. The Krn and KrnP2 modules contain the physical interrupt processing routines.*

Virtual Interrupt Processing

A virtual IRQ, or VIRQ, is useful with devices in Multi-Pak expansion slots. Because of the absence of an IRQ line from the Multi-Pak interface, these devices cannot initiate physical interrupts. VIRQ enables these devices to act as if they were interrupt drive. Use VIRQ only with device driver and pseudo device driver modules. VIRQ is handled in the **Clock** module, which handles the VIRQ polling table and installs the **`F$VIRQ`** system call. Since the **`F$VIRQ`** system call is dependent on clock initialization, the SysGo module forces the clock to start.

The virtual interrupt is set up so that a device can be interrupted at a given number of clock ticks. The interrupt can occur one time, or can be repeated as long as the device is used.

The **F$VIRQ** system call installs VIRQ in a table. This call requires specification of a 5-byte packet for use in the VIRQ table. This packet contains:

- Bytes for an actual counter
- A reset value for the counter
- A status byte that indicates whether a virtual interrupt has occurred and whether the VIRQ is to be reinstalled in the table after being issued

F$VIRQ also specifies an initial tick count for the interrupt. The actual call is summarized here and is described in detail in Chapter 8.

Call:	os9 F$VIRQ
Input:	(Y) = address of 5-byte packet
	(X) = 0 to delete entry, 1 to install entry
	(D) = initial count value
Output:	None
	(CC) carry set on error
	(IS) appropriate error code

The 5-byte packet is defined as follows:

Name	Offset	Function
Vi.Cnt	$0	Actual counter
Vi.Rst	$2	Reset value for counter
Vi.Stat	$4	Status byte

Two of the bits in the status byte are used. These are:

Bit 0 – set if a VIRQ occurs

Bit 7 – set if a count reset is required

When making an **F$VIRQ** call, the packet might require initialization with a reset value. Bit 7 of the status byte must be either set or cleared to signify a reset of the counter or a one-time VIRQ call. The reset value does not need to be the same as the initial counter value. When NitrOS-9 processes the call, it writes the packet address into the VIRQ table.

At each clock tick, NitrOS-9 scans the VIRQ table and subtracts one from each timer value. When a timer count reaches zero, NitrOS-9 performs the following actions:

1. Sets bit 0 in the status byte. This specifies a Virtual IRQ.

2. Checks bit 7 of the status byte for a count reset request.

3. If bit 7 is set, resets the count using the reset value. If bit 7 is reset, deletes the packet address from the VIRQ table.

When a counter reaches zero and makes a virtual interrupt request, NitrOS-9 runs the standard interrupt polling routine and services the interrupt. Because of this, you must install entries on both the VIRQ and IRQ polling tables whenever you are using a VIRQ.

Unless the device has an actual physical interrupt, install the device on the IRQ polling table via the **F$IRQ** system call before placing it on the VIRQ table.

If the device has a physical interrupt, use the interrupt's hardware register address as the polling address for the **F$IRQ** call. After setting the polling address, set the flip and mask bytes for the device and make the F$IRQ call.

If the device is totally VIRQ-driven, and has no interrupts, use the status byte from the VIRQ packet as the status byte. Use a mask byte of %00000001, defined as Vi.IFlag in the os9defs file. Use a flip byte value of 0.

See the appendix for example code using the VIRQ feature of NitrOS-9.

Chapter 3. Memory Modules

In Chapter 2, you learned that NitrOS-9 is based on the concept that memory is modular. This means that each program is considered to be an individually named object.

You also learned that each program loaded into memory must be in the module format. This format lets NitrOS-9 manage the logical contents of memory, as well as the physical contents. Module types and formats are discussed in detail in this chapter.

Module Types

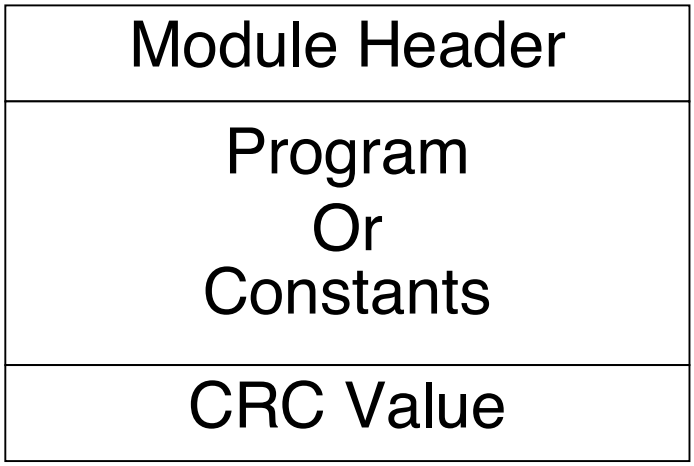
There are several types of modules. Each has a different use and function. These are the main requirements of a module:

- It cannot modify itself.
- It must be position-independent so that NitrOS-9 can load or relocate it wherever space is available. In this respect, the module format is the NitrOS-9 equivalent of *load records* used in older operating systems.

A module need not be a complete program or even 6809 machine language. It can contain BASIC09 I-code, constants, single subroutines, and subroutine packages.

Module Format

Each module has three parts: a *module header*, a *module body*, and a *cyclic-redundancy-check value* (CRC value).



Module Header

At the beginning of the module (the lowest address) is the module header. Its form depends upon the module's use. The header contains information about the module and its use. This information includes the following:

- Size
- Type (machine code, BASIC09 compiled code, and so on)
- Attributes (executable, re-entrant, and so on)
- Data storage memory requirements
- Execution starting address

Usually, you do not need to write routines to generate the modules and headers. All OS-9 programming languages automatically create modules and headers.

Module Body

The module body contains the program or constants. It usually is pure code. The module name string is included in this area.

The following figure provides the offset values for calculating the location of a module's name. (See "Offset to Module Name.")

CRC Value

The last three bytes of the module are the Cyclic Redundancy Check (CRC) value. The CRC value is used to verify the integrity of a module.

When the system first loads the module into memory, it performs a 24-bit CRC over the entire module, from the first byte of the module header to the byte immediately before the CRC. The CRC polynomial used is $800FE3.

As with the header, you usually don't need to write routines to generate the CRC value. Most OS-9 programs do this automatically.

Module Headers: Standard Information

The first nine bytes of all module headers are defined as follows:

Relative Address	Use
$00,$01	Sync bytes ($87,$CD)
$02,$03	Module size
$04,$05	Offset to module name
$06	Module type/language
$07	Attributes/revision level
$08	Header check

Sync Bytes

The sync bytes specify the location of the module. (The first sync byte is the start of the module.) These two bytes are constant.

Module Size

The module size specifies the size of the module in bytes (includes CRC).

Offset to Module Name

The offset to module name specifies the address of the module name string relative to the start of the module. The name string can be located anywhere in the module. It consists of a string of ASCII characters with the most significant bit set on the last character.

Type/Language Byte

The type/language byte specifies the type and language of the module.

The four most significant bits of this byte indicate the type. Eight types are predefined. Some of these are for OS-9's internal use only. The type codes are given here (0 is not a legal type code):

Code	Module Type	Name
$1x	Program module	Prgrm
$2x	Subroutine module	Sbrtn
$3x	Multi-module (for future use)	Multi
$4x	Data module	Data
$5x-$Bx	User-definable module	
$Cx	NitrOS-9 system module	Systm
$Dx	NitrOS-9 file manager module	FlMgr
$Ex	NitrOS-9 device driver module	Drivr
$Fx	NitrOS-9 device descriptor module	Devic

The four least significant bits of Byte 6 indicate the language (denoted by x in the previous Figure). The language codes are given here:

Code	Language
$x0	Data (non executable)
$x1	6809 object code
$x2	Basic09 I-Code
$x3	Pascal P-Code
$x4-$xF	Reserved for future use

By checking the language type, high-level language runtime systems can verify that a module is the correct type before attempting execution. Basic09, for example, can run either I-Code or 6809 machine language procedures arbitrarily by checking the language type code.

Attributes/Revision Level Byte

The attributes/revision level byte defines the attributes and revision level of the module.

The four most significant bits of this byte are reserved for module attributes. Currently, only Bit 7 is defined. When set, it indicates the module is re-entrant and, therefore, *shareable*.

The four least significant bits of this byte are a revision level in the range 0 to 15. If two or more modules have the same name, type, language, and so on, NitrOS-9 keeps in the module directory only the module having the highest revision level. Therefore, you can replace or patch a ROM module, simply by loading a new, equivalent module that has a higher revision level.

Note: *A previously linked module cannot be replaced until its link count goes to zero.*

Header Check

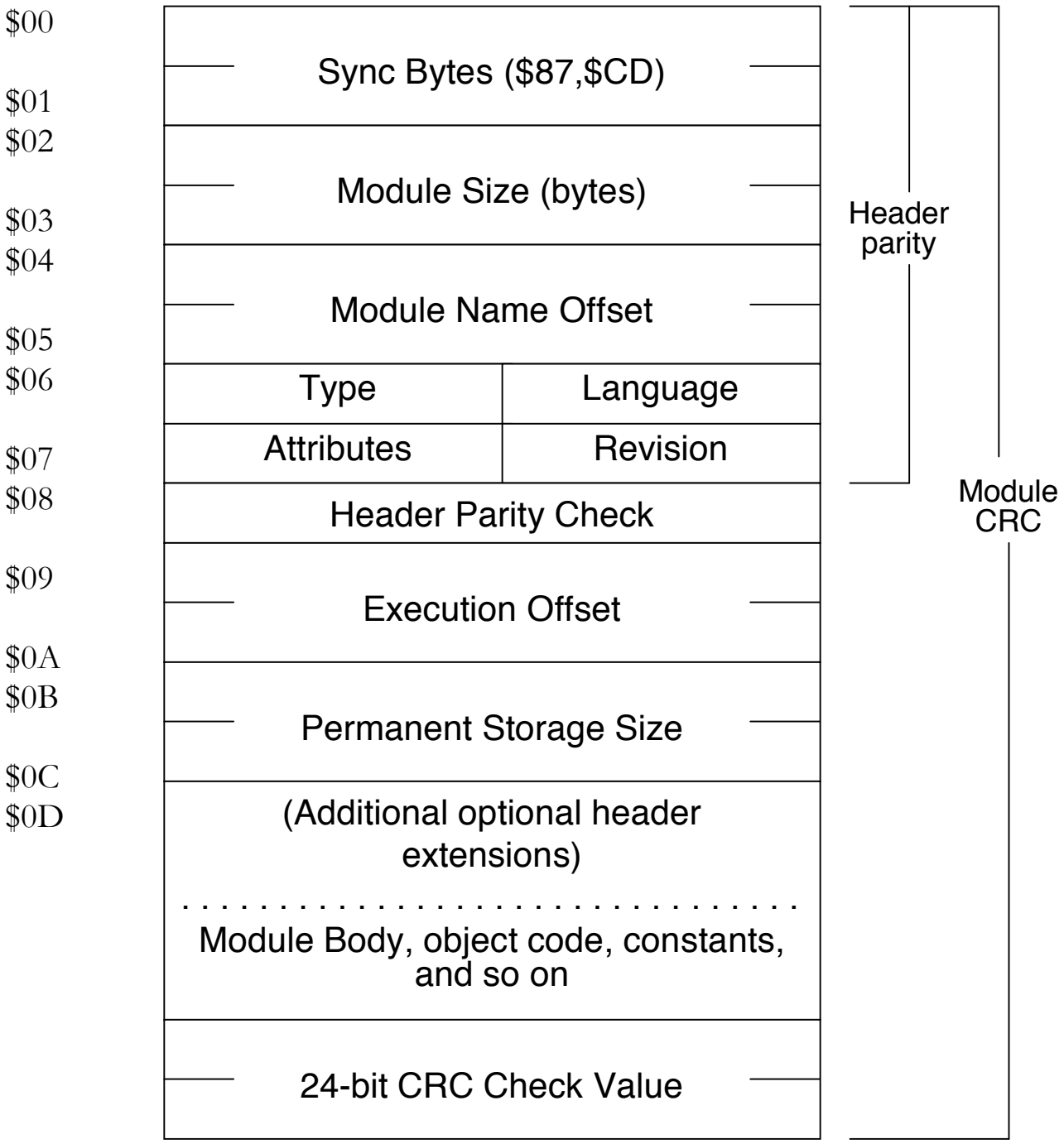
The header check byte contains the one's complement of the Exclusive-OR of the previous eight bytes.

Module Headers: Type-Dependent Information

More information usually follows the first nine bytes of a module header. The layout and meaning vary, depending on the module type.

Module types $Cx-$Fx (system module, file manager module, device driver module, and device descriptor module) are used only by OS-9. Their formats are given later in the manual.

Module types $Ix through $Bx have a general-purpose executable format. This format is often used in programs called by **`F$Fork`** or **`F$Chain`**. Here is the format used by these module types:


$00
$01
$02
$03
$04
$05
$06
$07
$08
$09
$0A
$0B
$0C
$0D
Sync Bytes ($87,$CD)
Module Size (bytes)
Module Name Offset
Type
Language
Attributes
Revision
Header Parity Check
Execution Offset
Permanent Storage Size
(Additional optional header extensions)
. . . . . .
Module Body, object code, constants, and so on
24-bit CRC Check Value
Header parity
Module CRC

As you can see from the preceding chart, the executable memory has four extra bytes in its header. They are:

$09,$0A	Execution offset
$0B,$0C	Permanent storage size

Execution Offset. The program or subroutine's offset starting address, relative to the first byte of the sync code. A module that has multiple entry points (such as cold start and warm start) might have a branch table starting at this address.

Permanent Storage Size. The minimum number of bytes of data storage required to run. Fork and Chain use this number to allocate a process' data area.

If the module is not directly executed by a Fork or Chain system call (for instance a subroutine package), this entry is not used by NitrOS-9. It is commonly used to specify the maximum stack size required by re-entrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

When NitrOS-9 starts after a single system reset, it searches the entire memory space for ROM modules. It finds them by looking for the module header sync code ($87,$CD).

When NitrOS-9 detects the header sync code, it checks to see if the header is correct. If it is, the system obtains the module size from the header and performs a 24-bit CRC over the entire module. If the CRC matches, NitrOS-9 considers the module to be valid and enters it into the module directory. All ROM modules that are present in the system at startup are automatically included in the system module directory.

After the module search, NitrOS-9 links to the component modules it found. This is the secret to NitrOS-9's ability to adapt to almost any 6809 computer. It automatically locates its required and optional component modules and rebuilds the system each time it is started.

Chapter 4. NitrOS-9's Unified Input/Output System

Chapter 1 mentioned that NitrOS-9 has a unified I/O system, consisting of all modules except those at the kernel level. This chapter discusses the I/O modules in detail.

The VDG interface performs both interface and low-level routines for VDG Color Computer 2 compatible modes and has limited support for high resolution screen allocation.

The GrfInt interface provides the standard code interpretations and interface functions.

The WindInt interface, available in the Multi-View package, contains all the functionality of GrfInt along with additional support features. If you use WindInt, do not include GrfInt.

Both WindInt and GrfInt use the low-level driver GrfDrv to perform drawing on the bitmap screens.

Term_VDG uses CC3I0 VDGINT while Term_win and all window descriptors use CC3I0/(WindInt/GrfInt)/GrfDrv modules.

The I/O system provides system-wide, hardware-independent I/O services for user programs and OS-9 itself. All I/O system calls are received by the kernel and passed to the I/O manager for processing.

The I/O manager performs some processing, such as the allocation of data structures for the I/O path. Then, it calls the file managers and device drivers to do most of the work. Additional file manager, device driver, and device descriptor

modules can be loaded into memory from files and used while the system is running.

The I/O Manager

The I/O manager provides the first level of service of I/O system calls. It routes data on I/O process paths to and from the appropriate file managers and device drivers.

The I/O Manager also maintains two important internal OS-9 data structures: the device table and the path table. Never modify the I/O manager.

When a path is opened, the I/O manager tries to link to a memory module that has the device name given or implied in the pathlist. This module is the device descriptor. It contains the names of the device driver and file manager for the device. The I/O manager saves the names so later system calls can be routed to these modules.

File Managers

NitrOS-9 can have any number of file manager modules. Each of these modules processes the raw data stream to or from a class of device drivers that have similar operational characteristics. It removes as many unique characteristics as possible from I/O operations. Thus, it assures that similar devices conform to the NitrOS-9 standard I/O and file structure.

The file manager also is responsible for mass storage allocation and directory processing, if these are applicable to the class of devices it serves. File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They can also monitor and process the data stream, for example, adding linefeed characters after carriage-return characters.

The file managers are re-entrant. The three standard NitrOS-9 file managers are:

- Random block file manager: The RBF manager supports random-access, block-structured devices such as disk systems and bubble memories. (Chapter 5 discusses the RBF manager in detail.)

- Sequential Character File Manager: The SCF manager supports single-character-oriented devices, such as CRTs or hardcopy terminals, printers, and modems. (Chapter 6 discusses SCF in detail.)

- Pipe File Manager (PIPEMAN): The pipe manager supports interprocess communication via pipes.

File Manager Structure

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

```
* All routines are entered with:
* (Y) = Path Descriptor pointer
* (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
```

Create, Open

Create and Open handle file creating and opening for devices. Typically, the process involves allocating any required buffers, initializing path descriptor variables, and establishing 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 Register B.

ChgDir

On multi-file devices, ChgDir searches for a directory file. If ChgDir finds the directory, it saves the address of the directory (up to four bytes) in the caller's process descriptor. The descriptor is located at P$DIO + 2 (for a data directory) or P$DIO + 8 (for an execution directory).

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

Delete

Multi-file device managers handle file delete requests by initiating a directory search that is similar to Open. Once a device manager finds the file, it removes the file from the directory.

Any media in use by the file are returned to unused status. In the case of the RBF manager, space is returned for system use and is marked as available in the free cluster bitmap on the disk. File managers that do not support multi-file devices return an error.

Seek

File managers that support random access devices use Seek to position file pointers of an already open path to the byte specified. Typically, the positioning is a logical movement. No error is produced at the time of the seek if the position is beyond the current "end of file."

Normally, file managers that do not support random access ignore Seek, However, an SCF-type manager can use Seek to perform cursor positioning.

Read

Read returns the number of bytes requested to the user's data buffer. Make sure Read returns 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 calls the device driver to actually read the data into the buffer. Then, the file manager copies the 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. 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. The RBF manager and similar random access devices that use fixed length records (sectors) must often pre-read a sector before writing it, unless they are writing the entire sector. In OS-9, writing past the end of file on a device expands the file with new data.

ReadLn

ReadLn differs from Read in two respects. First, ReadLn terminates when the first end-of-line (carriage return) is encountered. ReadLn performs any input editing that is appropriate for the device. In the case of SCF, editing involves handling functions such as backspace, line deletion, and the removal of the high order bit from characters.

WriteLn

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

GetStat, PutStat

The GetStat (get status) and PutStat (put status) system calls are wildcard calls designed to provide a method of accessing features of a device (or file manager) that are not generally device independent. The file manager can perform specific functions such as setting the size of a file to a given value. Pass unknown status calls to the driver to provide further means of device independence. For example, a PutStat call to format a disk track might behave differently on different types of disk controllers.

Close

Close is responsible for ensuring that any output to a device is completed. (If necessary, Close writes out the last buffer.) It releases any buffer space allocated in an Open or Create. Close does not execute the device driver's terminate

routine, but can do specific end-of-file processing if you want it to, such as writing end-of-file records on disks, or form feeds on printers.

Interfacing with Device Drivers

Strictly speaking, device drivers must conform to the general format presented in this manual. The I/O Manager is slightly different because it only uses 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. The Read entry point of an RBF driver, on the other hand, expects Read to return an entire sector.

The following code is part of an SCF file manager. The code shows how a file manager might call a driver.

```
********************
* IOEXEC
*   Execute Device's Read/Write Routine
*
* Passed:    (A) = Output character (write)
*            (X) = Device Table entry ptr
*            (Y) = Path Descriptor pointer
*            (U) = Offset of routine (D$Read,D$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  ,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
```

Device Driver Modules

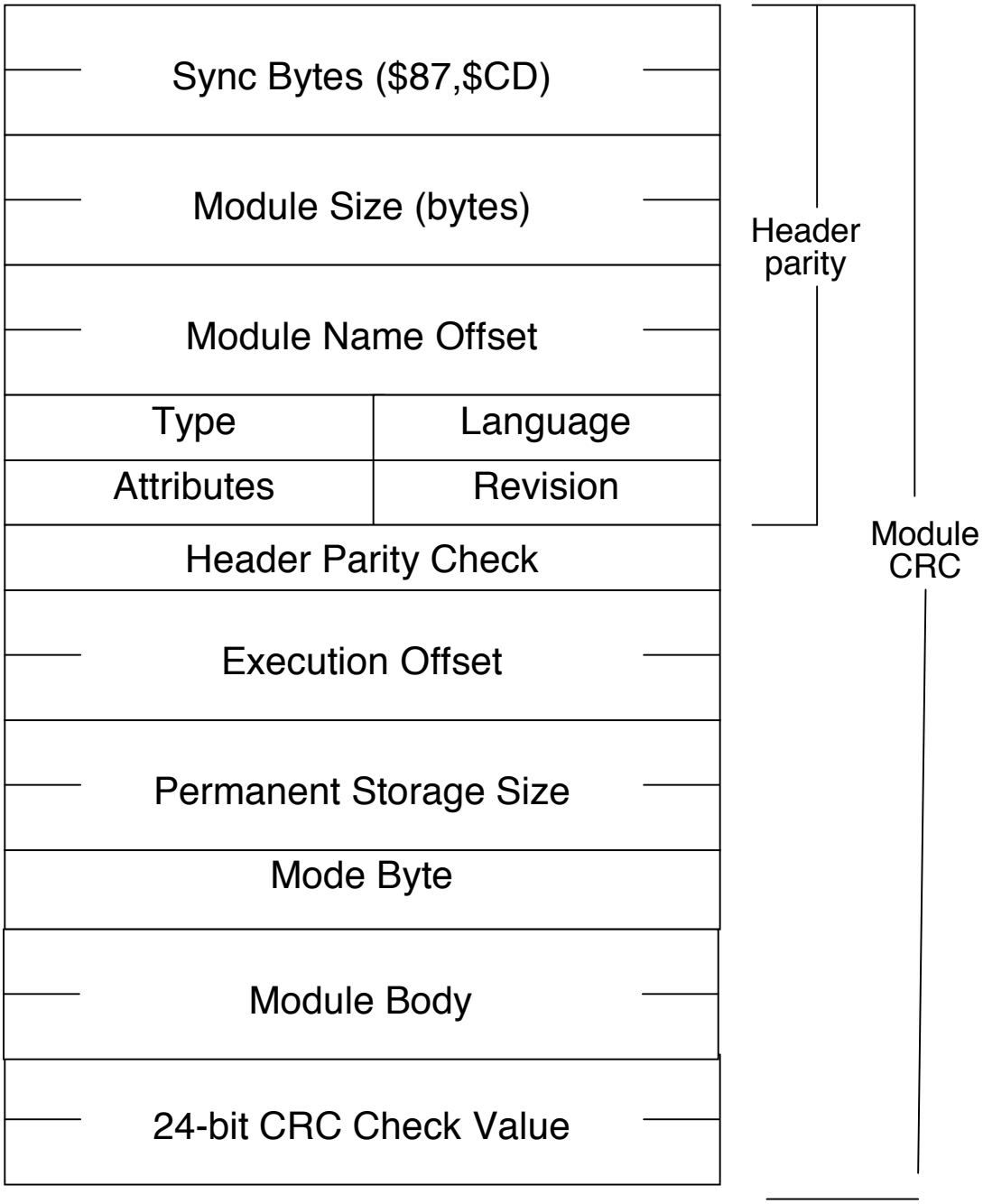
The device driver modules are subroutine packages that perform basic, low-level I/O transfers to or from a specific type of I/O device hardware controller. These modules are re-entrant. So, one copy of the module can concurrently run several devices that use identical I/O controllers.

Device driver modules use a standard module header, in which the module type is specified as code $Ex (device driver). The execution offset address in the module header points to a branch table that has a minimum of six 3-byte entries.

Each entry is typically an 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 passing the hardware control register address in the 6809 registers. The branch table looks like this:

Code	Meaning
$00	Device initialization routine
$03	Read form device
$06	Write to device
$09	Get device status
$0C	Set device status
$0F	Device termination routine

(For a complete description of the parameters passed to these subroutines, see the “Device Driver Subroutines” sections in Chapters 5 and 6.)


Sync Bytes ($87,$CD)
Module Size (bytes)
Module Name Offset
Type
Language
Attributes
Revision
Header Parity Check
Execution Offset
Permanent Storage Size
Mode Byte
Module Body
24-bit CRC Check Value
Header parity
Module CRC

NitrOS-9 Interaction with Devices

Device drivers often must wait for hardware to complete a task or for a user to enter data. Such a wait situation occurs if an SCF device driver receives a Read but there is no data is available, or if it receives a Write and no buffer space is available. NitrOS-9 drivers that encounter this situation should suspend the current process (via F$Sleep). In this way the driver allows other processes to continue using CPU time.

The most efficient way for a driver to awaken itself and resume processing data is by using interrupt requests (IRQs). It is possible for the driver to sleep for a number of system clock ticks and then check the device or buffer for a ready signal. The drawbacks to this technique are:

- It requires the system clock to always remain active.

- It might require a large number of ticks (perhaps 20) for the device to become ready. Such a case leaves you with a dilemma. If you make the program sleep for two ticks, the system wastes CPU time while checking for device ready. If the driver sleeps 20 ticks, it does not have a good response time.

An interrupt system allows the hardware to report to the CPU and the device drivers when the device is finished with an operation. Using interrupts to its advantage, a device driver can setup interrupt handling to occur when a character is sent or received or when a disk operation is complete. There is a built-in polling facility for pausing and awakening processes. Here is a technique for handling interrupts in a device driver:

1. Use the Init routine to place the driver interrupt service call (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 routine
os9 F$IRQ add dev to poll Sequence
bcs Error abnormal exit if error
```

2. Ensure that driver programs waiting for their hardware call the sleep routine. The sleep routine copies V.Busy to V.Wake. Then, it goes to sleep for some period of time.

3. When the driver program wakes up, have it check to see whether it was awakened by an interrupt or by a signal sent from some other process.

   Usually, the driver performs this check by reading the V.Wake storage byte. The V.Busy byte is maintained by the file manager to be used as 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 that there is a process awaiting an interrupt. The value in the Wake byte indicates the process to be awakened by sending a wakeup signal as shown in the following code:

```
    lda V.Busy,u get proc ID
    sta V.Wake,u arrange for wakeup
    andcc #^IntMasks prep for interrupts
Sleep50 ldx #0 or any other tick time (if signal test )
    OS9 F$Sleep await an IRQ
    ldx D.Proc get proc desc ptr if signal test
    ldb P$Signal,x i5 signal present? (if signal test)
    bne SigTest bra if 50 if Signal test
    tst V.Wake,u IRQ occur?
    bne Sleep50 bra if not
```

   Note that the code labeled “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 and 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 whether it is sending a wakeup signal or not.

4. When the device issues an interrupt, NitrOS-9 calls the routine at the address given in F$IRQ with the interrupts masked. Make the routine as short as possible, and have it return with an RTS instruction. IRQSERVC can verify that an interrupt has occurred for the device. It needs to clear the

interrupt to retrieve any data in the device. Then the V.Wake byte communicates with the main driver module. If V.Wake is non-zero, clear it to indicate a true device interrupt and use its contents as the process ID for an **F$Send** system call. The **F$Send** call sends a wakeup signal to the process. Here is an example:

```
     ldx V.Port,u get device address
     tst ?? is it real interrupt from device?
     bne IRQSVC90 bra to error if not
     lda Data,x get data from device
     sta 0,y
     lda V.Wake,u
     beq IRQSVC80 bra if none
     clr V.Wake,u clear it as flag to main routine
     ldb #S$Wake,u get wakeup signal
     os9 F$Send Send Signal to driver
IRQSVC80 clrb clear carry bit (all is well)
     rts
IROSVC90 comb Set carry bit (is an IRQ call)
     rts
```

Suspend State (NitrOS-9 Level 2 only)

The Suspend State allows the elimination of the F$Send system call during interrupt handling. Because 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 wake up the suspended main driver by clearing the process status byte suspend bit in the process state. Following are sample routines for the Sleep and IRQSERVC calls:

```
     lda D.Proc get process ptr
     sta V.Wake,u prep for re-awakening
     enable device to IRQ, give command, etc.
     bra Cmd5O 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 that D.Proc is a pointer to the process descriptor of the current process. Process 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 that a process is indeed suspended. In the other instance, it is a pointer to the process descriptor which enables the IRQSERVC routine to 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. Making the interrupts ensure that the IRQSERVC routine does not think it has cleared the suspend bit before it is even set. If this happens, when the bit is set the process might 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 P$State,x get state flag
     anda # Suspend clear suspend state
     sta P$State,x save it
     clrb clear carry bit
     rts
IRQSVCER comb Set carry bit
     rts
```

Device Descriptor Modules

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

Unlike the device drivers and file managers, which operate on classes of devices, each device descriptor tailors its functions to a specific device. Each device must have a device descriptor.

Device descriptor modules use a standard module header, in which the module type is specified as code $Fx (device descriptor). The name of the module is the name by which the system and user know the device (the device name given in path lists).

The rest of the device descriptor header consists of the information in the following chart:

Relative Address(es)	**Use**
$09,$OA	The relative address of the file manager name string address
$OB,$OC	The relative address of the device driver name string
$OD	Mode/Capabilities: D S PE PW PR E W R (directory, single user, public execute, public write, public read, execute, write, read)
$OE,$OF,$10	The absolute physical (24-bit) address of the device controller
$11	The number of bytes (n bytes) in the initialization table
$12,$12 + n	Initialization table

When OS-9 opens a path to the device, the system copies the initialization table into the option section (PD.OPT) of the path descriptor. (See “Path Descriptors” in this chapter.)

The values in this table can be used to define the operating parameters that are alterable by the Get Status and Set Status system calls (I$GetStt and I$SetStt). For example, parameters that are used when initializing terminals define which control characters are to be used for functions such as backspace and delete.

The initialization table can be a maximum of 32 bytes long. If the table is fewer than 32 bytes long, OS-9 sets the remaining values in the path descriptor to 0.

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

The following diagram illustrates the device descriptor format:

Device Descriptor Format

Name	**Relative Address**	**Bytes**	**Use**
M$ID	$00-$01	2	Sync Bytes ($87CD)
M$Size	$02-$03	2	Module Size (bytes)
M$Name	$04-$05	2	Offset to Module Name
M$Type	$06	1	Type / Language
M$Revs	$07	1	Attributes / Revision Level
M$Parity	$08	1	Header Parity Check
M$FMgr	$09-$0A	2	File Manager Name Offset
M$PDev	$0B-$0C	2	Device Driver Name Offset
M$Mode	$0D	1	Mode
M$Port	$0E-$10	3	Port Address
M$Opt	$11	1	Initialization Table Size
	$12,$12…n	n	Initialization table
			Name Strings, and so on
			CRC Check Value

Path Descriptors

Every open path is represented by a data structure called a path descriptor (PD). The PD contains the information the file managers and device drivers require to perform I/O functions.

PDs are 64 bytes long and are dynamically allocated and deallo cated by the I/O manager as paths are opened and closed.

They are internal data structures that are not normally referenced from user or applications programs. The description of PDs is presented here mainly for those

programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections. The first section, which is ten bytes long, is the same for all file managers and device drivers. The information in the first section is shown in the following chart.

Path Descriptor: Standard Information

Name	Address	Bytes	Use
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open paths using this PD
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)
PD.FST	$OA	22	Defined by the file manager
PD.OPT	$20	32	Reserved for the GetStat/SetStat options

PD.FST is 12-byte storage reserved for and defined by each type of file manager for file pointers, permanent variables, and so on.

PD.OPT is a 32-byte option area used for file or device operating parameters that are dynamically alterable. When the path is opened, the I/O manager initializes these variables by copying the initialization table that is in the device descriptor module. User programs can change the values later, using the Get Status and Set Status system calls.

PD.FST and **PD.OPT** are defined for the file manager in the assembly-language equate file (SCFDefs for the SCF manager or RBFDefs for the RBF manager).

Chapter 5. Random Block File Manager

The random block file manager (RBF manager) supports *disk storage*. It is a re-entrant subroutine package called by the I/O manager for I/O system calls to random-access devices. It maintains the logical and physical file structures.

During normal operation, the RBF manager requests allocation and deallocation of 256-byte data buffers. Usually, one buffer is required for each open file. When physical I/O functions are necessary, the RBF manager directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks (pages).

The RBF manager does not deal directly with physical addresses such as tracks and cylinders. Instead, it passes to the device drivers address parameters, using a standard address called a *logical sector number*, or *LSN*. LSNs are integers from 0 to *n*-1, where *n* is the maximum number of sectors on the media. The driver translates the logical sector number to actual cylinder/track/sector values.

Because the RBF manager supports many devices that have different performance and storage capacities, it is highly parameter-driven. They physical parameters it uses are stored on the media itself.

On disk systems, the parameters are written on the first few sectors of Track 0. The device drivers also use the information, particularly the physical parameters stored on Sector 0. These parameters are written by the FORMAT program that initializes and tests the disk.

Logical and Physical Disk Organization

All disks used by NitrOS-9 store basic information, file structure, and storage allocation information on these first few sectors.

LSN 0 is the *identification sector*. LSN 1 is the *disk allocation map sector*. LSN 2 marks the beginning of the disk's root directory. The following section tells more about LSN 0 and LSN 1.

Identification Sector (LSN 0)

LSN 0 contains a description of the physical and logical characteristics of the disk. These characteristics are set by the FORMAT command program when the disk is initialized.

The following table gives the NitrOS-9 mnemonic name, byte address, size, and description of each value stored in this LSN 0.

Name	Relative Address	Size (Bytes)	Use
DD.TOT	$00	3	Number of sectors on disk
DD.TKS	$03	1	Track size (in sectors)
DD.MAP	$04	2	Number of bytes in the allocation bit map
DD.BIT	$06	2	Number of sectors per cluster
DD.DIR	$08	3	Starting sector of the root directory
DD.OWN	$0B	2	Owner's user number
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 the bootstrap file
DD.BSZ	$18	2	Size of the bootstrap file (in bytes)
DD.DAT	$1A	5	Time of creation (Y:M:D:H:M)
DD.NAM	$1F	32	Volume name in which the last character has the most significant bit set
DD.OPT	$3F		Path descriptor options

Disk Allocation Map Sector (LSN 1)

LSN 1 contains the *disk allocation map*, which is created by FORMAT. This map shows which sectors are allocated to the files and which are free for future use.

Each bit in the allocation map represents a sector or cluster of sectors on the disk. If the bit is set, the sector is considered to be in use, defective, or non-existent. If the bit is cleared, the corresponding cluster is available. The allocation map usually starts at LSN 1. The number of sectors it requires varies according to how many bits are needed for the map. DD.MAP specifies the actual number of bytes used in the map.

Multiple sector allocation maps allow the number of sectors/cluster to be as small as possible for high volume media.

The FORMAT utility bases the size of the allocation map on the size and number of sectors per cluster.

The DD.MAP value in LSN 0 specifies the number of bytes (in LSN 1) that are used in the map.

Each bit in the disk allocation map corresponds to one sector cluster on the disk. The DD.BIT value in LSN 0 specifies the number of sectors per cluster. The number is an integral power of 2 (1, 2, 4, 8, 16, and so on).

If a cluster is available, the corresponding bit is cleared. If it is allocated, non-existent, or physically defective, the corresponding bit is set.

Root Directory

The file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as /D1). Usually, it immediately follows the Allocation Map. The location of the root directory file descriptor is specified in DD.DIR. The root directory contains an entry for each file that resides in the directory, including other directories.

File Descriptor Sector

The first sector of every file is the *file descriptor*. It contains the logical and physical description of the file.

The following table describes the contents of the file descriptor.

Name	Relative Address	Size (Bytes)	Use
FD.ATT	$00	1	File attributes: D S PE PW PR E W R (see next chart)
FD.OWN	$01	2	Owner's user ID
FD.DAT	$03	5	Date last modified (Y M D H M)
FD.LNK	$08	1	Link count
FD.SIZ	$09	4	File size (number of bytes)
FD.CREAT	$0D	3	Date created (Y M D)
FD.SEG	$10	240	Segment list (see next chart)

FD.ATT. The attribute byte contains the file permission bits. When set the bits indicate the following

Bit 7 Directory
Bit 6 Single user
Bit 5 Public execute
Bit 4 Public write
Bit 3 Public read
Bit 2 Execute
Bit 1 Write
Bit 0 Read

FD.SEG. The segment list consists of a maximum of 48 5-byte entries that have the size and address of each file block in logical order. Each entry has the block's 3-byte LSN and 2-byte size (in sectors). The entry following the last segment is zero.

After creation, a file has no data segments allocated to it until the first write. (Write operations past the current end-of-file cause sectors to be added to the file. The first write is always past the end-of-file.)

If the file has no segments, it is given an initial segment. Usually, this segment has the number of sectors specified by the minimum allocation entry in the device descriptor. If, however, the number of sectors requested is more than the minimum, the initial segment has the requested number.

Later expansions of the file usually are also made in minimum allocation increments. Whenever possible, NitrOS-9 expands the last segment instead of adding a segment. When the file is closed, NitrOS-9 truncates unused sectors in the last segment.

NitrOS-9 tries to minimize the number of storage segments used in a file. In fact, many files have only one segment. In such cases, no extra read operations are needed to randomly access any byte in the file.

If a file is repeatedly closed, opened, and expanded, it can become fragmented so that it has many segments. You can avoid this fragmentation by writing a byte at the highest address you want to be used on a file. Do this before writing any other data.

Directory

Disk directories are files that have the D attribute set. A directory contains an integral number of entries, each of which can hold the name and LSN of a file or another directory.

Each directory entry contains 29 bytes for the filename followed by three bytes for the LSN of the file's descriptor sector. The filename is left-justified in the field with the most significant bit of the last character set. Unused entries have a zero byte in the first filename character position.

Every disk has a master directory called the root directory. The DD.DIR value in LSN 0 (identification sector) specifies the starting sector of the root directory.

The RBF Manager Definitions of the Path Descriptor

As stated earlier in this chapter, the PD.FST section of the path descriptor is reserved for and defined by the file manager. The following table describes the use of this section by the RBF manager. For your convenience, it also includes the other sections of the PD.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers and device drivers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode 1 = read 2 = write 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
The RBF manager Path Descriptor Definitions (PD.FST Section)			
PD.SMF	$0A	1	State flag: Bit 0 = current buffer is altered Bit 1 = current sector is in the buffer Bit 2 = descriptor sector is in the buffer
PD.CP	$0B	4	Current logical file position (byte address)
PD.SIZ	$0F	4	File size
PD.SBL	$13	3	Segment beginning logical sector number (LSN)
PD.SBP	$16	3	Segment beginning physical sector number (PSN)
PD.SSZ	$19	3	Segment size
PD.DSK	$1C	2	Disk ID (for internal use only)
PD.DTB	$1E	2	Address of drive table

Name	Relative Address	Size (Bytes)	Use
The RBF manager Option Section Definitions (PD.OPT Section) (Copied from the device descriptor)			
PD.DTP	$20	1	Device class 0 = SCF 1 = RBF 2 = PIPE 3 = SBF

PD.DRV	$21	1	Drive number (0..*n*)
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 per track
PD.T0S	$2B	2	Default number of sectors per track (Track 0)
PD.ILV	$2D	1	Sector interleave factor
PD.SAS	$2E	1	Segment allocation size
PD.TFM	$2F	1	DMA transfer mode
PD.EXTEN	$30	2	Path extension for record locking
PD.STOFF	$32	1	Sector/track offsets
(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
PD.DFD	$37	3	Directory file descriptor PSN
PD.DCP	$3A	4	File's directory entry pointer
PD.DVT	$3E	2	Address of the device table entry

Any values not determined by this table default to zero.

RBF-Type Device Descriptor Modules

This section describes the use of the initialization table contained in the device descriptor modules for RBF-type devices. The following values are those the I/O manager copies from the device descriptor to the path descriptor.

Name	**Relative Address**	**Size (Bytes)**	**Use**
	$00-$11		Standard device descriptor module header
IT.DTP	$12	1	Device type: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
IT.DRV	$13	1	Drive number
IT.STP	$14	1	Step rate

IT.TYP	$15	1	Device type (see RBF path descriptor)
IT.DNS	$16	1	Media density: Always 1 (double) (see following information)
IT.CYL	$17	2	Number of cylinders (tracks)
IT.SID	$19	1	Number of sides
IT.VFY	$1A	1	0 = Verify disk writes 1 = no verify
IT.SCT	$1B	2	Default number of sectors per track
IT.T0S	$1D	2	Default number of sectors per track (Track 0)
IT.ILV	$1F	1	Sector interleave factor
IT.SAS	$20	1	Minimum size of segment allocation (number of sectors to be allocated at one time)

IT.DRV is used to associate a 1-byte integer with each drive that a controller handles. Number the drives for each controller as 0 to *n*-1, where *n* is the maximum number of drives the controller can handle.

IT.TYP specifies the device type (all types).

Bit 0 0 = 5-inch floppy diskette
Bit 5 0 = Non-Color Computer format
1 = Color Computer format
Bit 6 0 = Standard NitrOS-9 format
1 = Non-standard format
Bit 7 0 = Floppy diskette
1 = Hard disk

IT.DNS specifies the density capabilities (floppy diskette only).

Bit 0 0 = Single-bit density (FM)
1 = Double-bit density (MFM)
Bit 1 0 = Single-track density (5-inch, 48 tracks per inch)
1 = Double-track density (5-inch, 96 tracks per inch)

IT.SAS specifies the minimum number of sectors allowed at one time.

RBF Record Locking

Record locking is a general term that refers to methods designed to preserve the integrity of files that can be accessed by more than one user or process. The

NitrOS-9 implementation of record locking is designed to be as invisible as possible. This means that existing programs do not have to be rewritten to take advantage of record locking facilities. You can usually write new programs without special concern for multi-user activity.

Record locking involves detecting and preventing conflicts during record access. Whenever a process modifies a record, the system locks out other procedures from accessing the file. It defers access to other procedures until it is safe for them to write to the record. The system does not lock records during reads; so, multiple processes can read the records at the same time.

Record Locking and Unlocking

To detect conflicts, NitrOS-9 must recognize when a record is being updated. The RBF manager provides true record locking on a byte basis. A typical record update sequence is:

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

When a file is opened in update mode, any read causes locking of the record being accessed. This happens because the RBF manager cannot determine in advance if the record is to be updated. The record stays locked until the next read, write, or close.

However, when a file is opened in the read or execute modes, the system does not lock accessed records because the records cannot be updated in these two modes.

A subtle but important problem exists for programs that interrogate a data base and occasionally update its data. If you neglect to release a record after accessing it, the record might be locked indefinitely. This problem is characteristic of record locking systems and you can avoid it with careful programming.

Only one potion of a file can be locked at a time. If an application requires more than one record to be locked, open multiple paths to the same file and lock the record accessed by each path. RBF notices that the same process owns both paths and keeps them from locking each other.

Non-Shareable Files

Sometimes (although rarely), you must create a file that can never be accessed by more than one user at a time. To lock the file, you set the single-user bit in the file's attribute byte. You can do this by using the proper option when the file is created, or later using the NitrOS-9 ATTR command. Once the single-user bit is set, only one use can open the file at a time. If other users attempt to open the file, Error 253 is returned. Note, however, that non-shareable means only one path can be opened to a file at one time. Do not allow two processes to concurrently access a non-shareable file through the same path.

More commonly, you need to declare a file as single-user only during the execution of a specific program. You can do this by opening the file with the single-user bit set. For example, suppose a process is sorting a file. With the file's single-user bit set, NitrOS-9 treats the file exactly as though it had a single-user attribute. If another process attempts to open the file, NitrOS-9 returns Error 253.

You can duplicate non-shareable files by using the I$Dup system call. This means that it can be inherited and therefore accessible to more than one process at a time. Single-user means only that the file can be opened only once.

End-of-File Lock

A special case of record locking occurs when a user reads or writes data at the end of a file, creating an *EOF Lock*. An EOF Lock keeps the end of the file locked until a process performs a read or write that it is not at the end of the file. It prevents problems that might otherwise occur when two users want to simultaneously extend a file. The EOF Lock is the only case in which a write call automatically causes portions of a file to be locked. An interesting and useful side effect of the EOF Lock function occurs if a program creates a file for sequential output. As soon as the program creates the file, EOF Lock is set and no other process can *pass* the writer in processing the file. For example, if an assembler redirects a listing to a disk file, and a spooler utility tries to print a line from the file it is written, record locking makes the spooler wait and stay at least one step behind the assembler.

Deadlock Detection

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

When a deadly embrace occurs, RBF returns a deadlock error (Error 254) to the process that caused NitrOS-9 to detect the deadlock. To avoid deadlocks, make sure that processes always access records of shared files in the same sequence.

When a deadlock error occurs, it is not sufficient for a program to retry the operation that caused the error. If all processes use this strategy, none can ever succeed. For any process to proceed, at least one must cancel operation to release control over a requesting segment.

RBF-Type Device Driver Modules

An RBF-type device driver module contains a package of subroutines that perform sector-oriented I/O to or from a specific hardware controller. Such a module is usually re-entrant. Because of this, one copy of one device driver module can simultaneously run several devices that use identical I/O controllers.

The I/O manager allocates a permanent memory area for each device driver. The size of the memory area is given in the device driver module header. The I/O manager and the RBF manager use some of this area. The device driver can use the rest in any manner. This area is used as follows:

The RBF Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use
V.PAGE	$00	1	Port extended address bits A20-A16
V.PORT	$01	2	Device base address (defined by the I/O manager)
V.LPRC	$03	1	ID of the last active process (not used by RBF device drivers)
V.BUSY	$04	1	ID of the current process using driver (defined by RBF) 0 = no current process
V.WAKE	$05	1	ID of the process waiting for I/O completion

			(defined by the device driver)
V.USER	$06	0	Beginning of file manager specific storage
V.NDRV	$06	1	Maximum number of drives the controller can use (defined by the device driver)
	$07	8	Reserved
DRVBEG	$0F	0	Beginning of the drive tables
TABLES	$0F	DRVMEM*N	Space for number of tables reserved (*n*)
FREE		0	Beginning of space available for driver

These values are defined in files in the DEFS directory on the Development Package disk.

TABLES. This area contains one table for each drive that the controller handles. (The RBF manager assumes that there are as many tables as indicated by V.NDRV.) Some time after the driver Init routine is called, the RBF manager issues a request for the driver to read LSN 0 from a drive table by copying the first part of LSN 0 (up to DD.SIZ) into the table. Following is the format of each drive table:

Name	**Relative Address**	**Size (Bytes)**	**Use**
DD.TOT	$00	3	Number of sectors
DD.TKS	$03	1	Track size (in sectors)
DD.MAP	$04	2	Number of bytes in the allocation bit map
DD.BIT	$06	2	Number of sectors per bit (cluster size)
DD.DIR	$08	3	Address (LSN) of the root directory
DD.OWN	$0B	2	Owner's user number
DD.ATT	$0D	1	Disk access attributes (D S PE PW PR E W R)
DD.DSK	$0F	2	Disk ID (a pseudo-random number used to detect diskette swaps)
DD.FMT	$10	1	Media format
DD.SPT	$11	2	Number of sectors per track. (Track 0 can use a different value specified by IT.T0S in the device descriptor.)
DD.RES	$13	2	Reserved for future use
DD.SIZ	$15	0	Minimum size of device descriptor
V.TRAK	$15	2	Number of the current track (the track that the

			head is on, and the track updated by the driver)
V.BMB	$17	1	Bit-map use flag: 0 = Bit map is not in use (Disk driver routines must not alter V.BMB)
V.FILEHD	$18	2	Open file list for this drive
V.DISKID	$1A	2	Disk ID
V.BMAPSZ	$1C	1	Size of bitmap
V.MAPSCT	$1D	1	Lowest reasonable bitmap sector
V.RESBIT	$1E	1	Reserved bitmap sector
V.SCTKOF	$1F	1	Sector/track byte
V.SCOFST	$20	1	Sector offset split from byte above
V.TKOFST	$22	4	Reserved for future use
DRVMEM	$26	.	Size of each drive table

The format attributes (DD.FMT) are these:

Bit 0 Number of sides
- 0 = Single-sided
- 1 = Double-sided

Bit 1 Density
- 0 = Single-density
- 1 = Double-density

Bit 2 Track density
- 0 = Single (48 tracks per inch)
- 1 = Double (96 tracks per inch)

RBF Device Driver Subroutines

Like all device driver modules, RBF device drivers use a standard executable memory module format.

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a long branch (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
```

Ensure that each subroutine exits with the C bit of the condition code register cleared if no error occurred. If an error occurs, set the C bit and return an appropriate error code in Register B.

The rest of this chapter describes the RBF device driver subroutines and their entry and exit conditions.

Init

Initializes a device and the device's memory area.

Entry Conditions

Y address of the device descriptor
U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- If you want NitrOS-9 to verify disk writes, use the Request Memory system call (F$SRqMem) to allocate a 256-byte buffer area in which a sector can be read back and verified after a write.
- You must initialize the device memory area. For floppy diskette controllers, initialization typically consists of:
  1. Initializing V.NDRV to the number of drives with which the controller works
  2. Initializing DD.TOT (in the drive table) to a non-zero value so that Sector 0 can be read or written
  3. Initializing V.TRAK to $FF so that the first seek finds Track 0
  4. Placing the IRQ service routing on the IRQ polling list, using the Set IRQ system call (F$IRQ)
  5. Initializing the device control registers (enabling interrupts if necessary)
- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These areas contain the 24-bit device address.) Ensure the driver initializes each drive table appropriately for the type of diskette that the driver expects to be used on the corresponding drive.

Read Reads a 256-byte sector from a disk and places it in a 256-byte sector buffer.

Entry Conditions

B MSB of the disk's LSN
X LSB of the disk's LSN
Y address of the path descriptor
U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- The following is a typical routine for using Read:

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. The driver goes to sleep and waits for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it is not clear, the driver goes back to sleep.

- Whenever you read LSN 0, you must copy the first part of this sector into the proper drive table. (Get the drive number from PD.DRV in the path descriptor.) The number of bytes to copy is in DD.SIZ. Use the drive number (PD.DRV) to compute the offset for the corresponding drive table as follows:

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

Write Writes a 256-byte sector buffer to a disk.

Entry Conditions

B MSB of the disk LSN
X LSB of the disk LSN
Y address of the path descriptor
U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- Following is a typical routine for using Write:

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. The driver then goes to sleep and waits for the I/O to complete. (The IRQ service routine sends the wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it is not, the driver goes back to sleep. If the disk controller cannot be interrupt-driven, it is necessary to perform a programmed I/O transfer.
6. If PF.VFY in the path descriptor is equal to zero, read the sector back in and verify that it is written correctly. Verification usually does not involve a comparison of all of the data bytes.

- If disk writes are to be verified, the Init routine must request the buffer in which to place the sector when it is read back. Do not copy LSN 0 into the drive table when reading it back for verification.
- Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the Read routine.

GetStats and SetStats

Reads or changes device's operating parameters.

Entry Conditions

U address of the device memory area
Y address of the path descriptor
A status code

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- Get/set the device's operating parameters (status) as specified for the Get Status and Set Status system calls. GetStat and SetStat are wild card calls.
- It might be necessary to examine or change the register stack that contains the values of the 6809 registers at the time of the call. The address of the register stack is in PD.RGS, which is located in the path descriptor. You can use the following offsets to access any value in the register stack:

Reg.	Relative Address	Size	6809 Register
R$CC	$00	1	Condition code register
R$D	$01	2	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program counter

- Register D overlays Registers A and B.

Term Terminate a device.

Entry Conditions

U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).
- Following is a typical routine for using Term:

1. Wait until any pending I/O is 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 Return System Memory system call (F$SRtMem).

IRQ Service Routine

Services device interrupts

Additional Information

- The IRQ Service routine sends a wakeup signal to the process indicated by the process ID in V.WAKE when the I/O is complete. It then clears V.WAKE as a flag to indicate to the main program that the IRQ has indeed occurred.
- When the IRQ Service routine finishes servicing an interrupt, it must clear the carry and exit with an RTS instruction.
- Although this routine is not included in the device driver module branch table and is not called directly by the RBF manager, it is a key routine in interrupt-driven drivers. Its function is to:

1. Service the device interrupts (receive data from device or send data to it). The IRQ Service routine puts its data into and gets its data from buffers that are defined in the device memory area.
2. Wake up a process that is waiting for I/O to be completed. To do this, the routine checks to see if there is a process ID in V.WAKE (if the bit is non-zero); if so, it sends a wakeup signal to that process.
3. If the device is ready to send more data, and the out buffer is empty, disable the device's *ready to transmit* interrupts.

Boot (Bootstrap Module)

Loads the boot file into RAM.

Entry Conditions

None

Exit Conditions

D size of the boot file (in bytes)
X address at which the boot file was loaded into memory
CC carry set on error
B *error code* (if any)

Additional Information

- The Boot module is not part of the disk driver. It is a separate module that is stored on the boot track of the system disk with Krn and REL.
- The bootstrap module contains one subroutine that loads the bootstrap file and related information into memory. It uses the standard executable module format with a module type of $C. The execution offset in the module header contains the offset to the entry point of this subroutine.
- The module gets the starting sector number and size of the OS9Boot file from LSN 0. NitrOS-9 allocates a memory area large enough for the Boot file. Then, it loads the Boot file into this memory area.
- Following is a typical routine for using Boot:

1. Read LSN 0 from the disk into a buffer area. The Boot module must pick its own buffer area. LSN 0 contains the values for DD.BT (the 24-bit LSN of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes).
2. Get the 24-bit LSN of the bootstrap file from DD.BT.
3. Get the size of the bootstrap file from DD.BSZ. The Boot module 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 NitrOS-9 Request System Memory system call (F$SRqMem) to request the memory area in which the Boot file is loaded.
5. Read the Boot file into this memory area.
6. Return the size of the Boot file and its location. Boot file is loaded.

Chapter 6. Sequential Character File Manager

The Sequential Character File Manager (SCFMAN) supports devices that operate on a character-by-character basis. These include terminals, printers, and modems.

SCF is a re-entrant subroutine package. The I/O manager calls the SCF manager for I/O system handling of sequential, character-oriented devices. The SCF manager includes the extensive I/O editing functions typical of line-oriented operations, such as:

- backspace
- line delete
- line repeat
- auto line feed
- screen pause
- return delay padding

The SCF-type device driver modules are CC3IO, PRINTER, and RS-232. They run the video display, printer, and serial ports respectively. See the *NitrOS-9 Commands* manual for additional Color Computer I/O devices.

SCF Line Editing Functions

The SCF manager supports two sets of read and write functions. I$Read and I$Write pass data with no modification. I$ReadLn and I$WritLn provide full line editing of device functions.

Read and Write

The Read and Write system calls to SCF-type devices correspond to the BASIC09 GET and PUT statements. While they perform little modification to the data they pass, they do filter out keyboard interrupt, keyboard terminate, and

pause characters. (Editing is disabled if the corresponding character in the path descriptor contains a zero.

Carriage returns are not followed by line feeds or nulls automatically, and the high order bits are passed as sent/received.

Read Line and Write Line

The Read Line and Write Line system calls to SCF-type devices correspond to the BASIC09 INPUT, PRINT, READ, and WRITE statements. They provide full line editing of all functions enabled for a particular device.

The system initializes I$ReadLn and I$WritLn functions when you first use a particular device. (NitrOS-9 copies the option table from the device descriptor table associated with the specific device.

Later, you can alter the calls—either from assembly-language programs (using the Get Status system call), or from the keyboard (using the TMODE command). All bytes transferred by I$ReadLn and I$WritLn have the high order bit cleared.

SCF Definitions of the Path Descriptor

The PD.FST and PD.OPT sections of the path descriptor are reserved for an used by the SCF file manager.

The following table describes the SCF manager's use of PD.FST and PD.OPT. For your convenience, the table also includes the other sections of the path descriptor.

The PD.OPT section contains the values that determine the line editing functions. It contains many device operating parameters that can be read or written by the Set Status or Get Status system call. Any values not set by this table default to zero.

> **Note**: You can disable most of the editing functions by setting the corresponding control character in the path descriptor to zero. You can use the Set Status system call or the TMODE command to do this. Or, you can go a step further by setting the corresponding control character value in the device descriptor module to zero.

To determine the default settings for a specific device, you can inspect the device descriptor.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Sale for all file managers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read 2 = write 3 = update
PD.CNT	$02	1	Number of open images (paths using this path descriptor)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
SCF Path Descriptor Definitions (PD.FST Section)			
PD.DV2	$0A	2	Device table address of the second (echo) device
PD.RAW	$0C	1	Edit flag: 0 = raw mode 1 = edit mode
PD.MAX	$0D	2	Read Line 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

Name	Relative Address	Size (Bytes)	Use
SCF Option Section Definition (PD.OPT Section) (Copied from the device descriptor)			
PD.DTP	$20	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
PD.UPC	$21	1	Case:

			0 = uppercase and lowercase 1 = uppercase only
PD.BSO	$22	1	Backspace: 0 = backspace 1 = backspace, space, and backspace
PD.DLO	$23	1	Delete: 0 = backspace over line 1 = carriage return, line feed
PD.EKO	$24	1	Echo: 0 = no echo 1 = echo
PD.ALF	$25	1	Auto line feed: 0 = no auto line feed 1 = auto line feed
PD.NUL	$26	1	End-of-line null count: *N* = number of nulls ($00) sent after each carriage return or carriage return and line feed (*n* = $00-$FF)
PD.PAU	$27	1	End of page pause: 0 = no pause 1 = pause
PD.PAG	$28	1	Number of lines per page
PD.BSP	$29	1	Backspace character
PD.DEL	$2A	1	Delete-line character
PD.EOR	$2B	1	End-of-record character (End-of-line character) Read only. Normally set to $0D 0 = Terminate read-line only at the end of the file
PD.EOF	$2C	1	End-of-file character (read only)
PD.RPR	$2D	1	Reprint-line character
PD.DUP	$2E	1	Duplicate-last-line character
PD.PSC	$2F	1	Pause character
PD.INT	$30	1	Keyboard-interrupt character
PD.QUT	$31	1	Keyboard-terminate character
PD.BSE	$32	1	Backspace-echo character
PD.OVF	$33	1	Line-overflow character (bell CTRL-G)
PD.PAR	$34	1	Device initialization value (parity)
PD.BAU	$35	1	Software settable baud rate
PD.D2P	$36	2	Offset to second device name string
PD.XON	$38	1	ACIA XON character

PD.XOFF	$39	1	ACIA XOFF character
PD.ERR	$3A	1	Most recent I/O error status
PD.TBL	$3B	2	Copy of device table address
PD.PLP	$3D	2	Path descriptor list pointer
PD.PST	$3F	1	Current path status

PD.EOF specifies the end-of-file character. If this is the first and only character that is input to the SCF device, SCF returns an end-of-file error on Read or ReadLn.

PD.PSC specifies the pause character, which suspends output to the device before the next end-of-record character. The pause character also deletes any type-ahead input for ReadLn.

PD.INT specifies the keyboard-interrupt character. When the character is received, the system sends a keyboard-terminate signal to the last user of a path. The character also terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code.

PD.QUT specifies the keyboard-terminate character. When this character is received, the system sends a keyboard-terminate signal to the last user of a path. The system also cancels the current I/O request (if any) by sending an error code identical to the keyboard interrupt signal code.

PD.PAR specifies the parity information for external serial devices.

PD.BAU specifies baud rate, word length, and stop bit information for serial devices.

PD.XON contains either the character used to enable transmission of characters or a null character that disables the use of XON.

PD.XOFF contains either the character used to disable transmission of characters or a null character that disables the use of XOFF.

SCF-Type Device Descriptor Modules

The following chart shows how the initialization table in the device descriptors is used for SCF-type devices. The values are those the I/O manager copies from the device descriptor to the path descriptor.

An SCF editing function is turned off if its corresponding value is set to zero. For example, if IT.EOF is set to zero, there is no end-of-file character.

Name	Relative Address	Size (Bytes)	Use
(header)	$00-$11		Standard device descriptor module header
IT.DVC	$12	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
IT.UPC	$13	1	Case: 0 = upper- and lowercase 1 = uppercase only
IT.BSO	$14	1	Backspace: 0 = backspace 1 = backspace, space, and backspace
IT.DLO	$15	1	Delete: 0 = backspace over line 1 = carriage return
IT.EKO	$16	1	Echo: 0 = echo off 1 = echo on
IT.ALF	$17	1	Auto line feed: 0 = auto line feed disabled 1 = auto line feed enabled
IT.NUL	$18	1	End-of-line null count
IT.PAU	$19	1	Pause: 0 = end-of-page pause disabled 1 = end-of-page pause enabled
IT.PAG	$1A	1	Number of lines per page
IT.BSP	$1B	1	Backspace character
IT.DEL	$1C	1	Delete-line character
IT.EOR	$1D	1	End-of-record character
IT.EOF	$1E	1	End-of-file character
IT.RPR	$1F	1	Reprint-line character

IT.DUP	$20	1	Duplicate-last-line character
IT.PSC	$21	1	Pause character
IT.INT	$22	1	Interrupt character
IT.QUT	$23	1	Quit character
IT.BSE	$24	1	Backspace echo character
IT.OVF	$25	1	Line-overflow character (bell)
IT.PAR	$26	1	Initialization value—used to initialize a device control register when a path is opened to it (parity)
IT.BAU	$27	1	Baud rate
IT.D2P	$28	2	Attached device name string offset
IT.XON	$2A	1	X-ON character
IT.XOFF	$2B	1	X-OFF character
IT.COL	$2C	1	Number of columns for display
IT.ROW	$2D	1	Number of rows for display
IT.WND	$2E	1	Window number
IT.VAL	$2F	1	Data in rest of descriptor is valid
IT.STY	$30	1	Window type
IT.CPX	$31	1	X cursor position
IT.CPY	$32	1	Y cursor position
IT.FGC	$33	1	Foreground color
IT.BGC	$34	1	Background color
IT.BDC	$35	1	Border color

SCF-Type Device Driver Modules

An SCF-type device driver module contains a package of subroutines that perform raw (unformatted) data I/O transfers to or from a specific hardware controller. Such a module is usually re-entrant so that one copy of the module can simultaneously run several devices that use identical I/O controllers. The I/O manager allocates a permanent memory area for each controller sharing the driver.

The size of the memory area is defined in the device driver module header. The I/O manager and SCF use some of the memory area. The device driver can use the rest in any way (typically as variables and buffers). Typically, the driver uses the area as follows:

Name	**Relative Address**	**Size (Bytes)**	**Use**

V.PAGE	$00	1	Port extended 24-bit address
V.PORT	$01	2	Device base address (defined by the I/O manager)
V.LPRC	$03	1	ID of the last active process
V.BUSY	$04	1	ID of the active process (defined by RBF): 0 = no active process
V.WAKE	$05	1	ID of the process to reawaken after the device completes I/O (defined by the device driver): 0 = no waiting process
V.USER	$06	0	Beginning of file manager specific storage
V.TYPE	$06	1	Device type or parity
V.LINE	$07	1	Lines left until the end of the page
V.PAUS	$08	1	Pause request: 0 = no pause requested
V.DEV2	$09	2	Attached device memory area
V.INTR	$0B	1	Interrupt character
V.QUIT	$0C	1	Quit character
V.PCHR	$0D	1	Pause character
V.ERR	$0E	1	Error accumulator
V.XON	$0F	1	XON character
V.XOFF	$10	1	XOFF character
V.KANJI	$11	1	Reserved
V.KBUF	$12	2	Reserved
V.MODADR	$14	2	Reserved
V.PDLHD	$16	2	Path descriptor list header
V.RSV	$18	5	Reserved
V.SCF	$1D	0	End of SCF memory requirements
FREE	$1D	0	Free for the device driver to use

V.LPRC contains the process ID of the last process to use the device. The IRQ service routing sends this process the proper signal if it receives a quit character or an interrupt character. V.LPRC is defined by SCF.

V.BUSY contains the process ID of the process that is using the device. (If the device is not being used, V.BUSY contains a zero.) The process ID is used by SCF to prevent more than one process from using the device at the same time. V.BUSY is defined by SCF.

SCF Device Driver Subroutines

Like all device drivers, SCF device drivers use a standard executable memory module format.

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically an LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY   LBRA    INIT    Initialize driver
        LBRA    READ    Read character
        LBRA    WRITE   Write character
        LBRA    GETSTA  Get status
        LBRA    SETSTA  Set status
        LBRA    TERM    Terminate device
```

If no error occurs, each subroutine exits with the C bit in the Condition Code register cleared. If an error occurs, each subroutine sets the C bit and returns an appropriate error code in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init

Initializes device control registers and enables interrupts if necessary.

Entry Conditions

Y address of the device descriptor
U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (V.PAGE and V.PORT contain the device address.) There is no need to initialize the part of the memory area used by the I/O manager and SCF.
- Follow these steps to use Init:

1. Initialize the device memory area.
2. Place the IRQ service routine on the IRQ polling list, use the Set IRQ system call (F$IRQ).
3. Initialize the device control registers.

Read Reads the next character from the input buffer.

Entry Conditions

Y address of the path descriptor
U address of the device memory area

Exit Conditions

A character read
CC carry set on error
B *error code* (if any)

Additional Information

- This is a step by step description of a Read operation:

1. Read gets the next character from the input buffer.
2. If no data is ready, Read copies its process ID from V.BUSY into V.WAKE. It then uses the Sleep system call to put itself to sleep.
3. Later, when Read receives data, the IRQ service routine leaves the data in a buffer. Then, the routine checks V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine sends a wakeup signal to the waiting process.

- Data buffers are not automatically allocated. If a buffer is used, it defines it in the device memory area.

Write Sends a character (places a data byte in an output buffer) and enables the device output interrupts.

Entry Conditions

A character to write
Y address of the path descriptor
U address of the device memory area

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- If the data buffer is full, Write copies its process ID from V.BUSY into V.WAKE. Write then puts itself to sleep.
- Later, when the IRQ service routine transmits a character and makes room for more data, it checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, the routine sends a wakeup signal to that process.
- Write must ensure that the IRQ service routine that starts it begins to place data in the buffer. After an interrupt is generated, the IRQ service routine continues to transmit data until the data buffer is empty. Then, it disables the device's ready-to-transmit interrupts.
- Data buffers are not allocated automatically. If a buffer is used, define it in the device memory area.

GetSta and SetSta

Gets/sets device operating parameters (status) as specified for the Get Status and Set Status system calls. GetSta and SetSta are wildcard calls.

Entry Conditions

A depends on the function code
Y address of the path descriptor
U address of the device memory area
Other registers depend on the function code.

Exit Conditions

CC carry set on error
B *error code* (if any)
Other registers depend on the function code

Additional Information

- Any codes not defined by the I/O manager or SCF are passed to the device driver.
- You might need to examine or change the register stack that contains the values of the 6809 registers at the time of the call. The address of the register stack can be found in PD.RGS, which is located in the path descriptor.
- You can use the following offsets to access any value in the register packet:

Reg.	Relative Address	Size	6809 Register
R$CC	$00	1	Condition code register
R$D	$01	2	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program counter
The function code is retrieved from R$B on the user stack.			

Term

Terminates a device. Term is called when a device is no longer in use (when the link count of the device descriptor module becomes zero).

Entry Conditions

U *pointer to the device memory area*

Exit Conditions

CC carry set on error
B *error code* (if any)

Additional Information

- To use Term:

1. Wait until the IRQ service routine empties the output buffer.
2. Disable the device interrupts.
3. Remove the device from the IRQ polling list.

- When Term closes the last path to a device, NitrOS-9 returns to the memory pool the memory that the device used. If the device has been attached to the system using the I$Attach system call, NitrOS-9 does not return the static storage for the driver until an I$Detach call is made to the device. Modules contained in the Boot file are never terminated, even if their link counts reach zero.

IRQ Service Routine

Receives device interrupts. When I/O is complete, the routine sends a wakeup signal to the process identified by the process ID in V.WAKE. The routine also clears V.WAKE as a flag to indicate to the main program that the IRQ has occurred.

Additional Information

- The IRQ Service Routine is not included in the device driver branch tables, and is not called directly by SCF. However, it is a key routine in device drivers.
- When the IRQ Service routine finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instructions.
- Here is a typical sequence of events that the IRQ Service Routing performs:

1. Service the device interrupts (receive data from the device or send data to it). Ensure this routine puts its data into and gets its data from buffers that are defined in the device memory area.
2. Wake up any process that is waiting for I/O to complete. To do this, the routine checks to see if there is a process ID in V.WAKE (a value other than zero); if so, it sends 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 storage area to a value other than zero. The address of the attached device memory area is in V.DEV2.
5. If a keyboard terminate or interrupt character is received, signal the process in V.LPRC (last known process) if any.

Chapter 7. The Pipe File Manager (PIPEMAN)

The Pipe file manager handles control or processes that use paths to pipes. Pipes allow concurrently executing processes to send each other data by using the output of one process (the writer) as input to a second process (the reader). The reader gets input from the standard input. The exclamation point (!) operator specifies that the input or output is from or to a pipe. The Pipe file manager allocates a 256-byte block and a path descriptor for data transfer. The Pipe file manager also determines which process has control of the pipe. The Pipe file manager has the standard file manager branch table at its entry point:

```
ENTRY   LBRA    Create
        LBRA    Open
        LBRA    MakDir
        LBRA    ChgDir
        LBRA    Delete
        LBRA    Seek
        LBRA    PRead
        LBRA    PWrite
        LBRA    PRdLn
        LBRA    PWrLn
        LBRA    GetStat
        LBRA    SetStat
        LBRA    Close
```

You cannot use MakDir, ChgDir, Delete, and Seek with pipes. If you try to do so, the system returns E$UNKSVC (unknown service request). GetStat and SetStat are also no-action service routines. They return without error.

Create and Open perform the same functions. They set up the 256-byte data exchange buffer and save several addresses in the path descriptor.

The Close request checks to see if any process is reading or writing through the pipe. If not, NitrOS-9 returns the buffer.

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

The ! operator tells the Shell that processes wish to communicate through a pipe. For example:

```
poc1 ! proc2
```

In this example, shell forks Proc1 with the standard output path to a pipe and forks Proc2 with the standard input path from a pipe.

Shell can also handle a series of processes using pipe. For example:

```
proc1 ! proc2 ! proc3 ! proc4
```

The following outline shows how to set up pipes between processes:

```
Open /pipe      save path in variable x
Dup path #1     save stdout in variable y
Close #1        make path available
Dup x           put pipe in stdout
                (Dup uses lowest available)
Fork proc1      fork process 1
Close #1        make path available
Dup y           restore stdout
Close y         make path available

Dup path #0     save stdin in Y
Close #0        make path available
Dup x           put pipe in stdin
Fork proc2      fork process 2
Close #0        make path available
Dup y           restore stdin
Close x         no longer needed
Close y         no longer needed
```

Example: The following example shows how an application can initiate another process with the stdin and stdout routed through a 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 stdin path available
Close #1         make stdout path available
Dup a            make pipe1 stdin
Dup b            make pipe2 stdout
Fork new process
Close #0         make stdin path available
Close #1         make stdout path available
Dup x            restore stdin
Dup y            restore stdout
Return a&b       return pipe path numbers to caller
```

Chapter 8. System Calls

System calls are used to communicate between the NitrOS-9 operating system and assembly-language programs. There are two major types of calls—I/O calls and function calls.

Function calls include user mode calls and system mode calls.

Each system call has a mnemonic name. Names of I/O calls start with I$. For example, the Change Directory call is I$ChgDir. Names of function calls start with F$. For example, the Allocate Bits call is F$AllBit. The names are defined in the assembler-input conditions equate file called OS9Defs.

System mode calls are privileged. You can execute them only while NitrOS-9 is in the system state (when it is processing another system call, executing a file manager or device driver, and so on).

System mode calls are included in this manual primarily for programmers writing device drivers and other system-level applications.

Calling Procedure

To execute any system calls, you need to use an SWI2 instruction:

1. Load the 6809 registers with any appropriate parameters.
2. Execute an SWI instruction, followed immediately by a constant byte, which is the request code. In the references in this chapter, the first line is the system call name (for example Close Path) and the second line contains the call's mnemonic name (for example I$Close), the software interrupt Code 2 (103F), and the call's request code (for example, 8F) in hexadecimal.
3. After NitrOS-9 processes the call, it returns any parameters in the 6809 registers. If an error occurs, the C bit of the condition code register is set and Register B contains the appropriate error code. This feature permits a

BCS or BCC instruction immediately following the system call to branch either if there is an error or if no error occurs.

As an example, here is the Close system call:

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

You can use the assembler's *OS9* directive to simplify the call, as follows:

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

The ASM assembler allows any combination of upper- or lowercase letters. The RMA assembler, included in the *OS-9 Level Two Development Pak*, is case sensitive. The names in this manual have been spelled with upper and lower case letters, matching the defs for RMA.

I/O System Calls

NitrOS-9's I/O calls are easier to use than many other systems' I/O calls. This is because the calling program does not have to allocate and set up *file control blocks*, *sector buffers*, and so on.

Instead, NitrOS-9 returns a 1-byte path number whenever a process opens a path to a file or device. Until the path is closed, you can use this path number in later I/O requests to identify the file or device.

In addition, NitrOS-9 allocates and maintains its own data structures; so, you need not deal with them.

System Call Descriptions

The rest of this chapter consists of the system call descriptions. At the top of each description is the system call name, followed by its mnemonic name, the SWI2 code, and the request code. Next are the call's entry and exit conditions, followed by additional information about the code where appropriate.

In the system call descriptions, registers not specified as entry or exit conditions are not altered. Strings passed as parameters are normally terminated with a space character and end-of-line character, or with Bit 7 of the last character set.

If an error occurs on a system call, the C bit of Register CC is set and Register B contains the *error code*. If no error occurs, the C bit is clear and Register B contains a value of zero.

User Mode System Calls Quick Reference

Following is a summary of the User Mode System Calls referenced in this chapter:

F$AllBit	Sets bits in an allocation bit map
F$Chain	Chains a process to a new module
F$CmpNam	Compares two names
F$CpyMem	Copies external memory
F$CRC	Generates a cyclic redundancy check
F$DelBit	Deallocates bits in an allocation bit map
F$Exit	Terminates a process
F$Fork	Starts a new process
F$GBlkMp	Gets a copy of a system block map
F$GPrDsc	Gets a copy of a process descriptor
F$Icpt	Set a signal intercept trap
F$ID	Returns a process ID
F$Link	Links to a memory module
F$Load	Loads a module from mass storage
F$Mem	Changes a process's data area size
F$NMLink	Links to a module; does not map the module into the user's address space
F$NMLoad	Loads a module but does not map it into the user's address space
F$PErr	Prints an error message
F$PrsNam	Parses a pathlist name
F$SchBit	Searches a bit map
F$Send	Sends a signal to a process
F$Sleep	Suspends a process
F$SPrior	Sets a process's priority
F$SSWI	Sets a software interrupt vector
F$STime	Sets a system time

F$SUser	Sets a user ID number
F$Time	Returns the current time
F$Unlink	Unlinks a module
F$Unload	Unlinks a module by name
F$Wait	Waits for a signal
I$Attach	Attaches to an I/O device
I$ChgDir	Changes a working directory
I$Close	Closes a path
I$Create	Creates a new file
I$Delete	Deletes a file
I$DeletX	Deletes a file from the execution directory
I$Detach	Detaches an I/O device
I$Dup	Duplicates a path
I$GetStt	Gets a device's status
I$MakDir	Creates a directory file
I$Open	Opens a path to an existing file
I$Read	Reads data from a device
I$ReadLn	Reads a line of data from a device
I$Seek	Positions a file pointer
I$SetStt	Sets a device's status
I$Write	Writes data to a device
I$WritLn	Writes a data line to a device

System Mode Calls Quick Reference

Following is a summary of the System Mode Calls referenced in this chapter:

F$Alarm	Sets up an alarm
F$All64	Allocates a 64-byte memory block
F$AllHRAM	Allocates high RAM
F$AllImg	Allocates image RAM blocks
F$AllPrc	Allocates a process descriptor
F$AllRAM	Allocates RAM blocks
F$AllTsk	Allocates a process task number
F$AProc	Enters active process queue
F$Boot	Performs a system bootstrap
F$BtMem	Performs a memory request bootstrap
F$ClrBlk	Clears the specified block of memory
F$DATLog	Converts a DAT block offset to a logical address
F$DelImg	Deallocates image RAM blocks

F$DelPrc	Deallocates a process descriptor
F$DelRAM	Deallocates RAM blocks
F$DelTsk	Deallocates a process task number
F$ELink	Links modules using a module directory entry
F$FModul	Finds a module directory entry
F$Find64	Finds a 64-byte memory block
F$FreeHB	Gets a free high block
F$FreeLB	Gets a free low block
F$GCMDir	Compacts module directory entries
F$GProcP	Gets a process's pointer
F$IODel	Deletes an I/O module
F$IOQu	Puts an entry into an I/O queue
F$IRQ	Makes an entry into IRQ polling table
F$LDABX	Loads Register A from 0,X in Task B
F$LDAXY	Loads A[X,[Y]]
F$LDDDXY	Loads D[D+X,[Y]]
F$MapBlk	Maps the specified blocks
F$Move	Moves data to a different address space
F$NProc	Starts the next process
F$RelTsk	Releases a task number
F$ResTsk	Reserves a task number
F$Ret64	Returns a 64-byte memory block
F$SetImg	Sets a process DAT image
F$SetTsk	Sets a process's task DAT registers
F$Slink	Performs a system link
F$SRqMem	Performs a system memory request
F$SRtMem	Performs a system memory return
F$SSvc	Installs a function request
F$STABX	Stores Register A at 0,X in Task B
F$VIRQ	Makes an entry in a virtual IRQ polling table
F$VModul	Validates a module

User System Calls

Allocate Bits

Sets bits in an allocation bit map

OS9 F$AllBit 103F 13

Entry Conditions

D *number of the first bit to set*
X *starting address of the allocation bit map*
Y *number of bits to set*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Bit numbers range from 0 to *n*-1, where *n* is the number of bits in the allocation bit map.
- **Warning**: Do not issue the Allocate Bits call with Register Y set to 0 (a bit count of 0).

Chain

OS9 F$Chain 103F 05

Loads and executes a new primary module without creating a new process

Entry Conditions

A *language/type code*
B *size of the area* (in pages); must be at ???
X *address of the module name or filename*
Y *parameter area size* (in pages); defaults to zero if not specified
U *starting address of the parameter area*; must be at least one page

Error Output

CC carry set on error
B *error code* (if any)

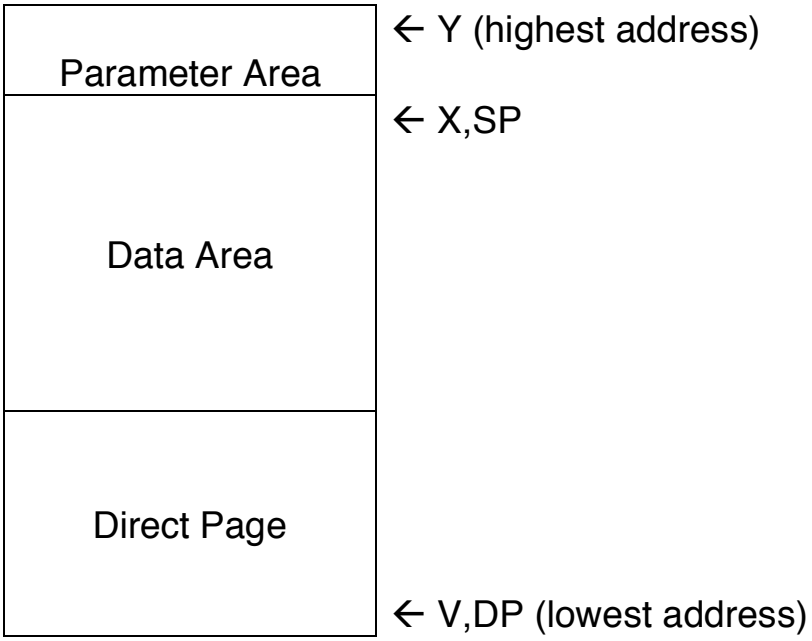
Additional Information

- Chain loads and executes a new primary module, but does not create a new process. A Chain system call is similar to a Fork followed by an Exit, but it has less processing overhead. Chain resets the calling process program and data memory areas and begins executing a new primary module. It does not affect open paths. This is a user mode system call.
- **Warning**: Make sure that the hardware stack pointer (Register SP) is located in the direct page before Chain executes. Otherwise the system might crash or return a suicide attempt error. This precaution also prevents a suicide in the event that the new module requires a smaller data area than that in use. Allow approximately 200 bytes of stack space for execution of the Chain system call.
- Chain performs the following steps:

1. It causes NitOS-9 to unlink the process's old primary module.
2. NitOS-9 parses the name string of the new process's primary module (the program that is to be executed first). Then, it causes NitrOS-9 to search the system module directory to see if a module with the same name, type, and language is already in memory.
3. If the module is in memory, it links to it. If the module is not in memory, it uses the name string as the pathlist of a file to load into memory. Then, it links to the first module in this file. (Several modules can be loaded from a single file.)

4. It reconfigures the data memory area to the size specified in the new primary module's header.
5. It intercepts and erases any pending signals.

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



   D *parameter area size*
   PC *module entry point absolute address*
   CC F=0, I=0; others are undefined

Registers Y and U (the top-of-memory and bottom-of-memory pointers, respectively) always have values at page boundaries. If the parent process does not specify a size for the parameter area, the size (Register D) defaults to zero. The data area must be at least one page.

(For more information, see the Fork system call.)

Compare Names

Compares two strings for a match

OS9 F$CmpNam 103F 11

Entry Conditions

B *length of string1*
X *address of string1*
Y *address of string2*

Exit Conditions

CC carry clear if the strings match

Additional Information

- The Compare Names call compares two strings and indicates whether they match. Use this call with the Parse Name system call. The second string must have the most significant bit (Bit 7) of the last character set.

Copy External Memory

Reads external memory into the user's buffer for inspection

OS9 F$CpyMem 103F 1B

Entry Conditions

D *DAT image pointer*
X *offset in block to begin copy*
Y *byte count*
U *caller's destination buffer*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- You can view any system memory through the use of the Copy External Memory call. The call assumes Register X is the address of the 64K address space described by the DAT image given.
- If you pass the entire DAT image of a process, place a value in Register X that equals the address in the process space. If you pass a partial DAT image (the upper half), place a value in Register X that equals the offset from the beginning of the DAT image ($8000).
- The support module for this call is KrnP2.

CRC

Calculates the CRC of a modle

OS9 F$CRC 103F 17

Entry Conditions

X *starting byte address*
Y *number of bytes*
U *address of the 3-byte CRC accumulator*

Exit Conditions

Updates the CRC accumulator.

Additional Information

- The CRC call calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators.
- The calculation begins at the *starting byte address* and continues over the specified *number of bytes*.
- You need not cover an entire module in one call since the CRC can be accumulated over several calls. The CRC accumulator can be any 3-byte memory area. You must initialize it to $FFFFFF before the first CRC call.
- The updated accumulator does not include the last three bytes of the module. The three CRC bytes are stored there.
- Be sure to initialize the CRC accumulator only once for each module check by CRC.

Deallocate Bits

Clears allocation map bits

OS9 F$DelBit 103F 14

Entry Conditions

D *number of the first bit to clear*
X *starting address of the allocation bit map*
Y *number of bits to clear*

Exit Conditions

None

Additional Information

- The Deallocate Bits call clears bits in the allocation bit map pointed to by Register X. Bit numbers are in the range 0 to *n*-1, where *n* is the number of bits in the allocation bit map.
- **Warning**: Do not call Deallocate Bits with Register Y set to zero (a bit count of zero).

Exit

Terminates the calling process

OS9 F$Exit 103F 06

Entry Conditions

B *status code to return to the parent process*

Exit Conditions

The process is terminated.

Additional Information

- The Exit system call is the only way a process can terminate itself. Exit deallocates the process's data memory area and unlinks the process's primary module. It also closes all open paths automatically.
- The Wait system call always returns to the parent the status code passed by the child in its Exit call. Therefore, if the parent executes a Wait and receives the status code, it knows the child has died.
- Exit unlinks only the primary module. Unlink any module that is loaded or linked to by the process before calling Exit.

Fork

Creates a child process

OS9 F$Fork 103F 03

Entry Conditions

A *language/type code*
B *size of the optional data area* (in pages)
X *address of the module name or filename* (See the following example)
Y *size of the parameter area* (in pages); defaults to zero if not specified
U *starting address of the parameter area*; must be at least one page

Exit Conditions

X *address of the last byte of the name* + 1 (See the following example)
A new process I/O number

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Fork creates a new process, a child of the calling process. Fork also sets up the child process's memory and 6809 registers and standard I/O paths.
- Before the Fork call:

T	E	S	T	$0D

□
X

- After the Fork call:

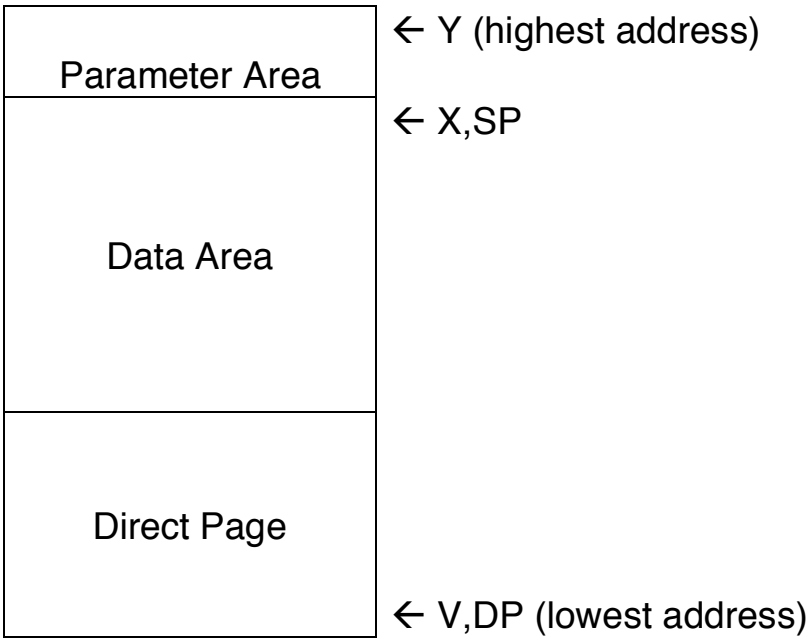
T	E	S	T	$0D

□
X

- This is the sequence of Fork's operations

1. NitOS-9 parses the name string of the new process's primary module (the program that NitrOS-9 executes first). Then, it searches the system module directory to see if the program already is in memory.
2. The next step depends on whether or not the program is already in memory. If the program is in memory, NitrOS-9 links the module to the process and executes it.

3. If the program is not in memory, NitrOS-9 uses the name as the pathlist of the file that is to be loaded into memory. Then, the first module in the this file is linked to and executed. (Several modules can be loaded from one file.)
4. NitrOS-9 uses the primary module's header to determine the initial size of the process's data area. It then tries to allocate a contiguous RAM area of that size. (This area includes the parameter passing area, which is copied from the parent process's data area.)
5. The new process's data memory area and registers are set up as show in the following diagram. NitrOS-9 uses the execution offset given in the module header to set the program counter to the module's entry point.



D *size of the parameter area*
PC *module entry point absolute address*
CC F=0, I=0, other condition code flags are undefined

Registers Y and U (the top-of-memory and bottom-of-memory pointers, respectively) always have values at page boundaries.

As stated earlier, if the parent does not specify the size of the parameter area, the size defaults to zero. The minimum overall data area size is one page.

When the shell processes a command line, it passes a string in the parameter area. The string is a copy of the parameter part of the command line. To simplify string-oriented processing, the shell also inserts an end-of-line character at the end of the parameter string.

Register X points to the start byte of the parameter string. If the command line includes the optional memory size specification (#*n* or #*n*K), the shell passes that size as the requested memory size when executing the Fork.

- If any of the preceding operations is unsuccessful, the Fork is terminated and NitrOS-9 returns an error to the caller.
- The child and parent processes execute at the same time unless the parent executes a Wait system call immediately after the Fork. In this case, the parent waits until the child dies before it resumes execution.
- Be careful when recursively calling a program that uses the Fork system call. Another child can be created with each new execution. This continues until the process table becomes full.
- Do not fork a process with a memory size of zero.

Get System Block Map

Gets a copy of the system block map

OS9 F$GBlkMp 103F 19

Entry Conditions

X *pointer to the 1024-byte buffer*

Exit Conditions

D *number of bytes per block* ($2000) (MMU block size dependent)
Y *system memory block map size*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Get System block Map call copies the system's memory block map into the user's buffer for inspection. The NitrOS-9 MFREE command uses this call to find out how much free memory exists.
- The support module for this call is KrnP2.

Get Module Directory

Gets a copy of the system module directory

OS9 F$GModDr 103F 1A

Entry Conditions

X *pointer to the 2048-byte buffer*
Y *end of copied module directory*
U *start address of the system module directory*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Get Module Directory call copies the system's module directory into the user's buffer for inspection. The NitrOS-9 MDIR command uses this call to read the module directory.
- The support module for this call is KrnP2.

Get Process Descriptor

Gets a copy of the process's process descriptor

OS9 F$GPrDsc 103F 18

Entry Conditions

A *requested process ID*
X *pointer to a 512-byte buffer*

Error Output

CC carry set on error
X *error code* (if any)

Additional Information

- The Get Process Descriptor call copies a process descriptor into the calling process's buffer for inspection. The data in the process descriptor cannot be changed. The NitrOS-9 PROCS command uses this call to get information about each existing process.
- The support module for this call is KrnP2.

Intercept

Sets a signal intercept trap

OS9 F$Icpt 103F 09

Entry Conditions

X *address of the intercept routine*
U *starting address of the routine's memory area*

Exit Conditions

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

Additional Information

- Intercept tells NitrOS-9 to set a signal intercept trap. Then, whenever the process receives a signal, NitrOS-9 executes the process's intercept routing.
- Store the address of the signal handler routine in Register X and the base address of the routine's storage area in Register U.
- Once the signal trap is set, NitrOS-9 can execute the intercept routine at any time because a signal can occur at any time.
- Terminate the intercept routine with an RTI instruction.
- If a process has not used the Intercept system call to set a signal trap, the process terminates if it receives a signal.
- This is the order in which F$Icpt operates:

1. When the process receives a signal, NitrOS-9 sets Registers U and B as follows:

   U *starting address of the intercept routine's memory area*
   B *signal code* (process's termination status)

   Note: The value of Register DP cannot be the same as it was when the Intercept call was made.
2. After setting the registers, NitrOS-9 transfers execution to the intercept routine.

Get ID

OS9 F$ID 103F 0C

Returns a caller's process ID and user ID

Entry Conditions

None

Exit Conditions

A *process ID*
Y *user ID*

Additional Information

- The *process ID* is a byte value in the range 1 to 255. NitrOS-9 assigns each process a unique process ID.
- The *user ID* is an integer from 0 to 65,535. It 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.
- On a single user system (such as the Color Computer 3), the user ID is inherited from CC3Go, which forks the initial shell.

Link

OS9 F$Link 103F 00

Links to a memory module that has the specified name, language, and type

Entry Conditions

A *type/language byte*
X *address of the module name* (See the following example)

Exit Conditions

A *type/language code*
B *attributes / revision level* (if no error)
X *address of the last byte of the module name* + 1 (See the following example)
U *module header absolute address*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The module's link count increases by one whenever Link references its name. Incrementing in this manner keeps track of how many processes are using the module.
- If the module requested is not shareable (not re-entrant), only one process can link to it at a time.
- Before the Fork call:

T	E	S	T	$0D

□
X

- After the Fork call:

T	E	S	T	$0D

□
X

- This is the order in which the Link call operates:

1. NitrOS-9 searches the module directory for a module that has the specified name, language, and type.

2. If NitrOS-9 finds the module, the address of the module's header is returned in Register U and the absolute address of the module's execution entry point is returned in Register Y. (This, and other information, is contained in the module header.)
3. If NitrOS-9 does not find the module or if the type/language codes in the entry and exit conditions do not math, NitrOS-9 returns one of the following errors

- Module not found
- Module busy (not shareable and in use)
- Incorrect or defective module header

Load

Loads a module or modules from a file

OS9 F$Load 103F 01

Entry Conditions

A *type / language code*; 0 = any language / type
X *address of the pathlist (filename)* (See the following example)

Exit Conditions

A *language / type code*
B *attributes / revision level* (if no error)
X *address of the last byte of the pathlist (filename)* + 1 (See the following example)
Y *primary module entry point address*
U *address of the module header*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Load call loads one or more modules from the file specified by a complete pathlist or from the working execution directory (if an incomplete pathlist is given).
- The file must have the execute access bit set. It also must contain one or more modules with proper module headers.
- NitrOS-9 adds all modules loaded to the system module directory. It links the first module read. The exit conditions apply only to the first module loaded.
- Before the Load call:

/	D	0	/	A	C	C	T	S	R	C	V	$0D

□
X

- After the Load call:

/	D	0	/	A	C	C	T	S	R	C	V	$0D

□
X

- Possible errors:
    - Module directory full
    - Memory full
    - Errors that occur on the Open, Read, Close, and Link system calls.

Memory

Changes process's data area size

OS9 F$Mem 103F 07

Entry Conditions

D *size of the new memory area* (in bytes);
0 = return current size and upper bound

Exit Conditions

Y *address of the new memory area upper bound*
D *actual size of the new memory* (in bytes)

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Memory call expands or contracts the process's data memory area to the specified size. Or, if you specify zero as the new size, the call returns the current size and upper boundaries of data memory.
- NitrOS-9 rounds off the size to the next page boundary. In allocating additional memory, NitrOS-9 continues upward from the previous highest address. In deallocating unneeded memory, it continues downward from that address.

Link to a module

OS9 F$NMLink 103F 21

Links to a module; does not map the module into the user's address space

Entry Conditions

A *type / language byte*
X *address of the module name*

Exit Conditions

A *type / language code*
B *module revision*
X *address of the last byte of the module name* + 1; any trailing blanks are skipped
Y *storage requirement for the module*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Although this call is similar to F$Link, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLink can therefore fork larger programs than can be forked by F$Link.

Load a module

OS9 F$NMLoad 103F 22

Loads one or more modules from a file but does not map the module into the user's address space

Entry Conditions

A *type / language byte*
X *address of the pathlist*

Exit Conditions

A *type / language code*
B *module revision*
X *address of the last byte of the pathlist* + 1
Y *storage requirement for the module*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- If you do not provide a full pathlist for this call, it attempts to load from a file in the current execution directory.
- Although this call is similar to F$Load, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLoad can therefore fork larger programs than can be forked by F$Load.

Print Error

OS9 F$PErr 103F 0F

Writes an error message to a specified path

Entry Conditions

B *error code*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Print Error writes an error message to the standard error path for the specified process. By default, NitrOS-9 shows:

  ERROR #decimal number

- The error reporting routine is vectored. Using the Set SVC system call, you can replace it with a more elaborate reporting module. To replace this routine use the Set SVC system call.

Parse Name

OS9 F$PrsNam 103F 10

Scans an input string for a valid NitrOS-9 name

Entry Conditions

X *address of the pathlist* (See the following example)

Exit Conditions

X *address of the optional slash* + 1
Y *address of the last character of the name* + 1
A *trailing byte* (delimiter character)
B *length of the name*

Error Output

CC carry set on error
B *error code* (if any)
Y *address of the first non-delimiter character in the string*

Additional Information

- Parses, or scans, the input text string for a legal NitrOS-9 name. It terminates the name with any character that is not a legal name character.
- Parse Name is useful for processing pathlist arguments passed to a new process.
- Because Parse Name processes only one name, you might need several calls to process a pathlist that has more than one name. As you can see from the following example, Parse Name finishes with Register Y in position for the next parse.
- If Register Y was at the end of a pathlist, Parse Name returns a bad name error. It then moves the pointer in Register Y past any space characters so that it can parse the next pathlist in a command line.
- Before the Parse Name call:

/	D	0	/	P	A	Y	R	O	L	L			

□
X

- After the Parse Name call:

/	D	0	/	P	A	Y	R	O	L	L			

□ □ B=2

X Y

Search Bits

OS9 F$SchBit 103F 12

Searches a specified memory allocation bit map for a free memory block of a specified size

Entry Conditions

D *starting bit number*
X *starting address of the map*
Y *bit count (free bit block size)*
U *ending address of the map*

Exit Conditions

D *starting bit number*
Y *bit count*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Search Bit call searches the specified allocation bit map for a free block (cleared bits) of the required length. The search starts at the *starting bit number*. If no block of the specified size exists, the call returns with the carry set, starting bit number, and size of the largest block.

Send

Sends a signal to a specified process

OS9 F$Send 103F 08

Entry Conditions

A *destination process ID*
B *signal code*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The *signal code* is a single byte value in the rage 0 through 255.
- If the destination process is sleeping or waiting, NitrOS-9 activates the process so that the process can process the signal.
- If a signal trap is set up, F$Send executes the signal processing routing (Intercept). If none was set up, the signal terminates the destination process and the signal code becomes the exit status. (See the Wait system call.) An exception is the wakeup signal; that signal does not cause the signal intercept routine to be executed.
- Signal codes are defined as follows:

0	System terminate (cannot be intercepted)
1	Wake up the process
2	Keyboard terminate
3	Keyboard interrupt
128-255	User defined

- If you try to send a signal to a process that has a signal pending, NitrOS-9 cancels the current Send call and returns an error. Issue a Sleep call for a few ticks; then, try again.
- The Sleep call saves CPU time. See the Intercept, Wait, and Sleep system calls for more information.

Sleep

Temporarily turns off the calling process

OS9 F$Sleep 103F 0A

Entry Conditions

X One of the following:
sleep time (in ticks)
0 = sleep indefinitely
1 = sleep for the remainder of the current time slice

Exit Conditions

X *sleep time minus the number of ticks that the process was asleep*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- If Register X contains zero, NitrOS-9 turns the process off until it receives a signal. Putting a process to sleep is a good way to wait for a signal or interrupt without wasting CPU time.
- If Register X contains one, NitrOS-9 turns the process off for the remainder of the process's current time slice. It inserts the process into the active process queue immediately. The process resumes execution when it reaches the front of the queue.
- If Register X contains an integer in the rage 2-255, NitrOS-9 turns off the process for the specified number of ticks, *n*. It inserts the process into the active process queue after *n*-1 ticks. The process resumes execution when it reaches the front of the queue. If the process receives a signal, it awakens before the time has elapsed.
- When you select processes among multiple windows, you might need to sleep for two ticks.

Set Priority

Changes the priority of a process

OS9 F$SPrior 103F 0D

Entry Conditions

A *process ID*
B *priority*
0 lowest
255 highest

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Set Priority changes the process's priority to the priority specified. A process can change another process's priority only if it has the same user ID.

Set SWI

OS9 F$SSWI 103F 0E

Sets the SWI, SWI2, and SWI3 vectors

Entry Conditions

A *SWI type code*
X *address of the user software interrupt routine*

Error Output

C carry set on error
B *error code* (if any)

Additional Information

- Sets the interrupt vectors for SWI, SWI2, and SWI3 instructions.
- Each process has its own local vectors. Each Set SWI call sets one type of vector according to the code number passed in Register A:

  1 SWI
  2 SWI2
  3 SWI3

- When NitrOS-9 creates a process, it initializes all three vectors with the address of the NitrOS-9 service call processor.
- **Warning**: Microware-supplied software uses SWI2 to call NitrOS-9. If you reset this vector, these programs cannot work. If you change all three vectors, you cannot call NitrOS-9 at all.

Set Time

Sets the system time and date

OS9 F$STime 103F 16

Entry Conditions

X *relative address of the time packet*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Set Time sets the current system date and time and starts the system real-time clock. The date and time are passed in a time packet as follows:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Then, the call makes a link system call to find the clock. If the link is successful, NitrOS-9 calls the clock initialization. The clock initialization:

1. Sets up hardware dependent functions
2. Sets up the F$Time system call via F$SSVc

Set User ID Number

OS9 F$SUser 103F 1C

Changes the current user ID without checking for errors or checking the ID number of the caller

Entry Conditions

Y *desired user ID number*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The support module for this call is Krn.

Time

Gets the system date and time

OS9 F$Time 103F 15

Entry Conditions

X *address of the area in which to store the date and time packet*

Exit Conditions

X *date and time*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- The Time call returns the current system date and time in the form of a 6-byte packet (in binary). NitrOS-9 copies the packet to the address passed in Register X.
- The packet looks like this:

- Time is a part of the clock module and it does not exist if no previous call to F$STime has been made.

Unlink

OS9 F$UnLink 103F 02

Unlinks (removes from memory) a module that is not in use and that has a link count of zero

Entry Conditions

U *address of the module header*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- Unlink unlinks a module from the current process's address space, decreases its link count by one, and, if the link count becomes zero, returns the memory where the module was located to the system for use by other processes.
- You cannot unlink system modules or device drivers that are in use.
- Unlink operates in the following order:

1. Unlink tells NitrOS-9 that the calling process no longer needs the module.
2. NitrOS-9 decreases the module's link count by one.
3. When the resulting link count is zero, NitrOS-9 destroys the module.

   If any other process is using the module, the module's link count cannot fall to zero. Therefore, NitrOS-9 does not destroy the module.

- If you pass a bad address, Unlink cannot find a module in the module directory and does not return an error.

Unlink a Module By Name

OS9 F$UnLoad 103F 1D

Decrements a specified module's link count, and removes the module from memory if the resulting link count is zero

Entry Conditions

A *module type*
X *pointer to module name*

Error Output

CC carry set on error
B *error code* (if any)

Additional Information

- This system call differs from Unlink in that it uses a pointer to the module name instead of the address of the module header.
- The support module for this call is KrnP2.

Wait

Temporarily turns off a calling process

OS9 F$Wait 103F 04

Entry Conditions

None

Exit Conditions

A *deceased child process's ID*
B *deceased child process's exist status* (if no error)

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Wait call turns off the calling process until a child process *dies*, either by exiting an Exit system call, or by receiving a signal. The Wait call helps you save system time.
- NitrOS-9 returns the child's process ID and exit status to the parent. If the child died because of a signal, the exit status byte (Register B) contains the signal code.
- If the caller has several children, NitrOS-9 activates the caller when the first one dies. Therefore, you need to use one Wait system call to detect the termination of each child.
- NitrOS-9 immediately reactivates the caller if a child dies before the Wait call. If the caller has no children, Wait returns an error. (See the Exit system call for more information.)
- If the Wait call returns with the carry bit set, the Wait function was not successful. If the carry bit is cleared, Wait functioned normally and any error that occurred in the child process is returned in Register B.

Attach

Attaches a device to the system or verifies device attachment.

OS9 I$Attach 103F 80

Entry Conditions

A *access mode*
X *address of the device name string*

Exit Conditions

X *updated past device name*
U *address of the device table entry*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Attach does not reserve the device. It only prepares the device for later use by any process.
- NitrOS-9 installs most devices automatically on startup. Therefore, you need to use Attach only when installing a device dynamically or when verifying the existence of a device. You need not use the Attach system call to perform routing I/O.
- The access mode parameter specifies the read and/or write operations to be allowed. These are:

  0 = Use any special device capabilities
  1 = Read only
  2 = Write only
  3 = Update (read and write)

- Attach operates in this sequence:

1. NitrOS-9 searches the system module to see if any memory contains a device descriptor that has the same name as the device.
2. NitrOS-9's next operation depends on whether or not the device is already attached. If NitrOS-9 finds the descriptor and if the device is not already attached, NitrOS-9 link the device's file manager and device driver. It then

places the address of the manager and the driver in a new device table entry. NitrOS-9 then allocates any memory needed by the device driver, and calls the driver's initialization routine, which initializes the hardware.

3. If NitrOS-9 finds the descriptor, and if the device is already attached, NitrOS-9 verifies the attachment.

Change Directory

OS9 I$ChgDir 103F 86

Changes the working directory of a process to a directory specified by a pathlist.

Entry Conditions

A *access mode*
X *address of the pathlist*

Exit Conditions

X *updated past pathlist*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- If the access mode is read, write, or update, NitrOS-9 changes the current data directory. If the access mode is execute, NitrOS-9 changes the current execution directory.
- The calling process must have read access to the directory specified (public read if the directory is not owned by the calling process).
- The access modes are:

  0 = Use any special device capabilities
  1 = Read only
  2 = Write only
  3 = Update (read and write)

Close Path

Terminates an I/O path

OS9 I$Close 103F 8F

Entry Conditions

A *path number*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Close Path terminates the I/O path to the file or device specified by *path number*. Until you use another Open, Dup, or Create system call for that path, you can no longer perform I/O to the file or device.
- If you close a path to a single-user device, the device becomes available to other requesting processes. NitrOS-9 deallocates internally managed buffers and descriptors.
- The Exit system call automatically closes all open paths. Therefore, you might not need to use the Close Path system call to close some paths.
- Do not close a standard I/O path unless you want to change the file or device to which it corresponds.
- Close Path performs an implied I$Detach call. If it causes the device link count to become 0, the device termination routine is executed. See I$Detach for additional information.

Create File

Creates and opens a disk file

OS9 I$Create 103F 83

Entry Conditions

A *access mode* (write or update)
B *file attributes*
X *address of the pathlist*

Exit Conditions

A *path number*
X *address of the last byte of the pathlist + 1; skips any trailing blanks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- NitrOS-9 parses the pathlist and enters the new filename in the specified directory. If you do not specify a directory, NitrOS-9 enters the new filename in the working directory.
- NitrOS-9 gives the file the attributes passed in Register B, which has bits defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Shareable file

- These access mode parameters passed in Register A must have the write bit set if any data is to be written. These access codes are defined as follows: 2 = write, 3 = update. The mode affects the file only until the file is closed.
- You can reopen the file in any access mode allowed by the file attributes. (See the Open system call.)

- Files opened for write can allow faster data transfer than those opened for update because update sometimes needs to pre-read sectors.
- If the execute bit (Bit 2) is set, the file is created in the working execution directory instead of the working data directory.
- Create File causes an implicit I$Attach call. If the device has not previously been attached, the device's initialization routine is called.
- Later I/O calls use the path number to identify the file, until the file is closed.
- NitrOS-9 does not allocate data storage for a file at creation. Instead, it allocates the storage either automatically when you first issue a write or explicitly by the SetStat subroutine.
- If the filename already exists in the directory, an error occurs. If the call specifies a non-multiple file device (such as a printer or terminal), Create behaves the same as Open.
- You cannot use Create to make directories. (See the Make Directory system call for instructions on how to make directories.)
- Before the Create File call:

/	D	0	/	W	O	R	K	$0D

□
X

- After the Create File call:

/	D	0	/	W	O	R	K	$0D

□
X

Delete File

Deletes a specified disk file

OS9 I$Delete 103F 87

Entry Conditions

X *address of the pathlist*

Exit Conditions

X *address of the last byte of the pathlist + 1; skips any trailing blanks*

Error Output

CC carry set on error

B *error code*, if any

Additional Information

- The Delete File call deletes the disk file specified by the pathlist. The file must have write permission attributes (public write, if the calling process is not the owner). An attempt to delete a device results in an error. The caller must have non-shareable write access to the file or an error results.

Example

Before the Delete File call:

/	D	0	/	W	O	R	K				M	E	M	O	$0d

X

After the Delete File Call

/	D	0	/	W	O	R	K				M	E	M	O	$0d

X

Delete A File

OS9 I$DeletX 103F 90

Deletes a file from the current data or current execution directory

Entry Conditions

A *access mode*
X *address of the pathlist*

Exit Conditions

X *address of the last byte of the pathlist + 1; skips any trailing blanks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Delete A File call removes the disk file specified by the selected pathlist. This function is similar to I$Delete except that it access an access mode byte. If the access mode is execute, the call selects the current execution directory. Otherwise, it selects the current data directory.
- If a complete pathlist is provided (the pathlist begins with a slash (/)), the access mode of the call ignores the access mode.
- Only use this call to delete a file. If you attempt to use I$DeletX to delete a device, the system returns an error.

Detach Device

Removes a device from the system device table

OS9 I$Detach 103F 81

Entry Conditions

U *address of the device table entry*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Detach Device call removes a device from both the system and the system device table, assuming the device is not being used by another process. You must use this call to detach devices attached using the Attach system call. Attach and Detach are both used mainly by the I/O manager. SCF also uses Attach and Detach to set up its second device (echo device).
- This is the sequence of the operation of Detach Device:

1. Detach Device calls the device driver's termination routine. Then, NitrOS-9 deallocates any memory assigned to the driver.
2. NitrOS-9 unlinks the associated device driver and file manager modules.
3. NitrOS-9 then removes the driver, as long as no other module is using that driver.

Duplicate Path

Returns a synonymous path number

OS9 I$Dup 103F 82

Entry Conditions

A *old path number* (number of path to duplicate)

Exit Conditions

A *new path number* (if no error)

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Duplicate Path returns another, synonymous path number for the file or device specified by the *old path number*.
- The shell uses the Duplicate Path call when it redirects I/O.
- System calls can use either path number (old or new) to operate on the same file or device.
- Makes sure that no more than one process is performing I/O on any one path at the same time. Concurrent I/O on the same path can cause unpredictable results with RBF files.
- The I$Dup call always uses the lowest available path number. This lets you manipulate standard I/O paths to contain any desired paths.

Get Status

Returns the status of a file or device

OS9 I$GetStt 103F 8D

Entry Conditions

A *path number*
B *function code*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Status is a *wildcard* call. Use it to handle device parameters that:
  - Are not the same for all devices
  - Are highly hardware-dependent
  - Must be user-changeable
- The exact operation of the Get Status system call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's parameters for such functions as backspace character and echo on/off. The Get Status call is commonly used with the Set Status call.
- The Get Status function codes that are currently defined are listed in the "Get Status System Calls" section.

Make Directory

Creates and initializes a directory

OS9 I$MakDir 103F 85

Entry Conditions

B *directory attributes*
X *address of the pathlist*

Exit Conditions

X *address of the last byte of the pathlist + 1; skips any trailing blanks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Make Directory call creates and initializes a directory as specified by the pathlist. The directory contains only two entries, one for itself (.) and one for its parent directory (..).
- NitrOS-9 makes the calling process the owner of the directory.
- Because the Make Directory call does not open the directory, it does not return a path number.
- The new directory automatically has its directory bit set in the access permission attributes. The remaining attributes are specified by the byte passed in Register B. The bits are defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Single-user
7	Don’t care

Open Path

OS9 I$Open 103F 84

Opens a path to an existing file or device as specified by the pathlist

Entry Conditions

A *access mode* (D S PE PW PR E W R)
X *address of the pathlist*

Exit Conditions

A *path number*
X *address of the last byte of the pathlist + 1; skips any trailing blanks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- NitrOS-9 searches for the file in one of the following:
  - The directory specified by the pathlist if the pathlist begins with a slash.
  - The working data directory, if the pathlist does not begin with a slash.
  - The working execution directory, if the pathlist does not begin with a slash and if the execution bit is set in the access mode.
- NitrOS-9 returns a path number for later system calls to use to identify the file.
- The access mode parameter lets you specify which read and/or write operations are to be permitted. When set, each access mode bit enables one of the following: Write, Read, Read and Write, Update, Directory I/O.
- The access mode must conform to the access permission attributes associated with the file or device. (See the Create system call.) Only the owner can access a file unless the appropriate public permission bits are set.
- The update mode might be slightly slower than the others because it might require pre-reading of sectors for random access of bytes within sectors.
- Several processes (users) can open files at the same time. Each device has an attribute that specifies whether or not it is shareable.
- If the single-user bit is set, the file opened for singer-user access regardless of the settings of the file's permission bits.

- You must open directory files for read or write if the directory bit (Bit 7) is set in the access mode.
- Open Path always uses the lowest path number available for the process.

Read

Read *n* bytes from a specified path

OS9 I$Read 103F 89

Entry Conditions

A *path number*
X *address in which to store the data*
Y *number of bytes to read*

Exit Conditions

Y *number of bytes read*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Read call reads the specified number of bytes from the specified path. It returns the data exactly as read from the file/device without additional processing or editing. The path must be opened in the read or update mode.
- If there is not enough data in the specified file to satisfy the read request, the read call reads fewer bytes than requested but an end-of-file error is not returned. After all data in file is read, the next I$Read call returns an end-of-file error.
- If the specified file is open for update, the record read is locked out on RBF-type devices.
- The keyboard terminate, keyboard interrupt, and end-of-file characters are filtered out of the Entry Conditions data on SCF-type devices unless the corresponding entries in the descriptor have been set to zero. You might want to modify the device descriptor so that these values are initialized to zero when the path is opened.
- The call reads the number of bytes requested unless Read encounters any of the following:
    - An end-of-file character
    - An end-of-record character (SCF only)
    - An error

Read Line With Editing

Reads a text line with editing

OS9 I$ReadLn 103F 8B

Entry Conditions

A *path number*
X *address at which to store data*
Y *maximum number of bytes to read*

Exit Conditions

Y *number of bytes read*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Read Line is similar to Read. The difference is that Read Line reads the input file or device until it encounters a carriage return character or until it reaches the maximum byte count specified, whichever comes first. The Read Line also automatically activates line editing on character oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.
- SCF requires that the last byte entered be an end-of-record character (usually a carriage return). If more data is entered than the maximum specified, Read Line does not accept it and a PD.OVF character (usually a bell) is echoed.
- After one Read Line call reads all the data in a file, the next Read Line call generates an end-of-file error.
- For more information about line editing, see “SCF Line Editing Functions” in Chapter 6.

Seek

Repositions a file pointer

OS9 I$Seek 103F 88

Entry Conditions

A *path number*
X *MS 16 bits of the desired file position*
Y *LS 16 bits of the desired file position*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Seek call repositions the path's logical file pointer, the 32-bit address of the next byte in the file to be read from or written to.
- You can perform a seek to any value, regardless of the file's size. Later writes automatically expand the file to the required size (if possible). Later reads, however, return an end-of-file condition. Note that a seek to Address 0 is the same as a rewind operation.
- NitrOS-9 usually ignores seeks to non-random access devices, and returns without error.
- 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.

Set Status

Sets the status of a file or device

OS9 I$SetStt 103F 8E

Entry Conditions

A *path number*
B *function code*
Other registers depend on the function code

Error Output

CC carry set on error
B *error code*, if any
Other registers depend on the function code

Additional Information

- Set Status is a wildcard call. Use it to handle device parameters that:
  - Are not the same for all devices
  - Are highly hardware-dependent
  - Must be user-changeable
- The exact operation of the Set Status system call depends on the device driver and file manager associated with the path. A typical use is to set a terminal's parameters for such functions as backspace character and echo on/off. The Set Status call is commonly used with the Get Status call.
- The Set Status function codes that are currently defined are listed in the "Set Status System Calls" section.

Write

Writes to a file or device

OS9 I$Write 103F 8A

Entry Conditions

A *path number*
X *starting address of data to write*
Y *number of bytes to write*

Exit Conditions

Y *number of bytes written*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Write system call writes to the file or device associated with the path number specified.
- Before using Write, be sure the path is opened or created in the write or update access mode. NitrOS-9 writes data to the file or device without processing or editing the data. NitrOS-9 automatically expands the file if you write data path the present end-of-file.

Write Line

Writes to a file or device until it encounters a carriage return

OS9 I$WritLn 103F 8C

Entry Conditions

A *path number*
X *address of the data to write*
Y *maximum number of bytes to read*

Exit Conditions

Y *number of bytes written*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Writes to the file or device that is associated with the path number specified.
- Write Line is similar to Write. The difference is that Write Line writes data until it encounters a carriage return character. It also activates line editing for character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.
- Before using Write Line, be sure the path opened or created in the write or update access mode.
- For more information about line editing, see “SCF Line Editing Functions” in Chapter 6.

Set an Alarm

OS9 F$Alarm 103F 1E

Sets an alarm to ring the bell at a specified time

Entry Conditions

X *relative address of time packet*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- When the system reaches the specified alarm time, it rings the bell for 15 seconds.
- The time packet is identical to the packet used in the F$STime call. See F$STime for additional information on the format of the packet.
- All alarms begin at the start of a minute and any seconds in the packet are ignored.
- The system is limited to one alarm at a time.

Allocate 64

OS9 F$All64 103F 30

Dynamically allocates 64-byte blocks of memory

Entry Conditions

X *base address of the page table*; 0 = the page table has not been allocated

Exit Conditions

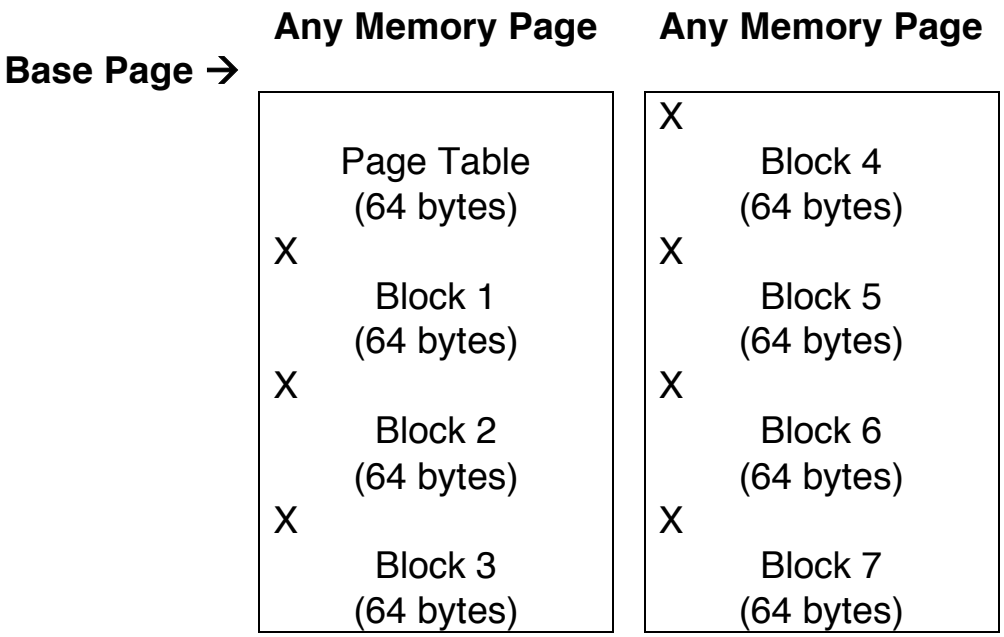
A *block number*
X *base address of the page table*
Y *address of the block*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Allocate 64 system call allocates the 64-byte blocks of memory by splitting pages (256-byte sections) into four sections.
- NitrOS-9 uses the first 64 bytes of the base page as a page table. This table contains the page number (most significant byte of the address) of all pages in the memory structure. If Register X passes a value of zero, the call allocates a new base page and the first 64-byte memory block.
- Whenever a new page is needed, a Request System Memory system call (F$SRqMem) executes automatically.
- The first byte of each block contains the block number. Routines that use the Allocate 64 call should not alter this byte.
- The following diagram shows how seven blocks might be allocated:


Any Memory Page
Any Memory Page
Base Page →
Page Table
(64 bytes)
X
Block 1
(64 bytes)
X
Block 2
(64 bytes)
X
Block 3
(64 bytes)
X
Block 4
(64 bytes)
X
Block 5
(64 bytes)
X
Block 6
(64 bytes)
X
Block 7
(64 bytes)

Allocate High RAM

Allocate system memory from high physical memory

OS9 F$AllHRam

Entry Conditions

B *number of blocks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This call searches for the desired number of contiguous free RAM blocks, starting its search at the top of memory. F$AllHRam is similar to F$AllRAM except F$AllRAM begins its search at the bottom of memory.
- Screen allocation routines use this call to provide a better chance of finding the necessary memory for a screen.

Allocate Image

OS9 F$AllImg 103F 3A

Allocates RAM blocks for process DAT image

Entry Conditions

A *starting block number*
B *number of blocks*
X *process descriptor pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Use the Allocate Image system call to allocate a data area for a process. The blocks that Allocate Image define might not be contiguous.
- The support module for this system call is Krn.

Allocate Process Descriptor

Allocates and initializes a 512-byte process descriptor

OS9 F$AllPrc 103F 4B

Entry Conditions

None

Exit Conditions

U *process descriptor pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The process descriptor table houses the address of the descriptor. Initialization of the process descriptor consists of clearing the first 256 bytes of the descriptor, setting up the state as a system state, and marking as unallocated as much of the DAT image as the system allows—typically, 60-64 kilobytes.
- The support module for this system call is KrnP2. The call also brances to the F$SRqMem call.

Allocate RAM

OS9 F$AllRAM 103F 39

Searches the memory block map for the desired number of contiguous free RAM blocks

Entry Conditions

B *number of blocks*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The support module for this system call is Krn.

Allocate Process Task Number

Determines whether NitrOS-9 has assigned a task number to the specified process

OS9 F$AllTsk 103F 3F

Entry Conditions

X *process descriptor pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- If the process does not have a task number, NitrOS-9 allocates a task number and copies the DAT image into the DAT hardware.
- The support module for this call is Krn. Allocate Process Task Number also branches to F$ResTsk and F$SetTsk.

Insert Process

Inserts a process into the queue for execution

OS9 F$AProc 103F 2C

Entry Conditions

X *address of the process descriptor*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Insert Process system call inserts a process into the active process queue so that NitrOS-9 can schedule the process for execution.
- NitrOS-9 sorts all processes in the queue by process age (the count of how many process switches have occurred since the process's last time slice). When a process is moved to the active process queue, NitrOS-9 sets its age according to its priority—the higher the priority, the higher the age.
- An exception is a newly active process that was deactivated while in the system state. NitrOS-9 gives such a process higher priority because the process usually is executing critical routines that affect shared system resources.

Bootstrap System

OS9 F$Boot 103F 35

Links either the module named Boot or the module specified in the INIT module

Entry Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- When it calls the linked module, Boot expects to receive a pointer giving it the location and size of an area in which to search for the new module.
- The support module for this call is Krn. Bootstrap System also branches to the F$Link and F$VModul system calls.

Bootstrap Memory Request

OS9 F$BtMem 103F 36

Allocates the requested memory (rounded to the nearest block) as contiguous memory in the system's address space

Entry Conditions

D *byte count requested*

Exit Conditions

D *byte count granted*
U *pointer to allocated memory*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This call is identical to F$SRqMem.
- The Bootstrap Memory Request support module is Krn.

Clear Specified Block

Marks blocks in the process DAT image as unallocated

OS9 F$ClrBlk 103F 50

Entry Conditions

B *number of blocks*
U *address of first block*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- After Clear Specified Block deallocates blocks, the blocks are free for the process to use for other data or program areas. If the block address passed to Clear Specified Block is invalid or if the call attempts to clear the stack area, it return E$IBA.
- The support module for the call is KrnP2.

DAT to Logical Address

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

OS9 F$DATLog 103F 44

Entry Conditions

B *DAT image offset*
X *block offset*

Exit Conditions

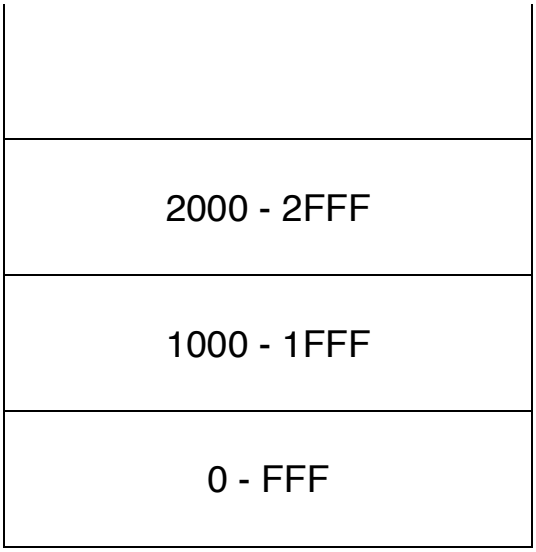
X *logical address*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Following is a sample conversion:



Input: B = 2
X = $0329

Output: X=$2329

- The support module for this call is Krn.

Deallocate Image RAM Blocks

Deallocates image RAM blocks

OS9 F$DelImg 103F 3B

Entry Conditions

A *number of starting block*
B *block count*
X *process descriptor pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This system call deallocates memory from a process's address space. It frees the RAM for system use and frees the DAT image for the process. Its main use is to let the system clean up after a process death.
- The support module for this call is KrnP2.

Deallocate Process Descriptor

Returns a process descriptor's memory to a free memory pool

OS9 F$DelPrc 103F 4C

Entry Conditions

A *process ID*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Use this call to clear the system scratch memory and stack area associated with the process.
- The support module for this call is KrnP2.

Deallocate RAM blocks

Clears a block's RAM In Use flag in the system memory block map

OS9 F$DelRAM 103F 51

Entry Conditions

B *number of blocks*
X *starting block number*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Deallocate RAM Blocks call assumes the blocks being deallocated are not associated with any DAT image.
- The support module for this call is KrnP2.

Deallocate Task Number

Releases the task number that the process specified by the passed descriptor pointer

OS9 F$DelTsk 103F 40

Entry Conditions

X *process descriptor pointer*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The support module for this call is Krn.

Link Using Module Directory Entry

Performs a link using a pointer to a module directory entry

OS9 F$ELink 103F 4D

Entry Conditions

B *module type*
X *pointer to module directory entry*

Exit Conditions

U *module header address*
Y *module entry point*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This call differs from Link in that you supply a pointer to the module directory entry rather than a pointer to a module name.
- The support module for this call is Krn.

Find Module Directory Entry

Returns a pointer to the module directory entry

OS9 F$FModul 103F 4E

Entry Conditions

A *module type*
X *pointer to the name string*
Y *DAT image pointer* (for name)

Exit Conditions

A *module type*
B *module revision number*
X *updated name string* (if Register A contains 0 on entry)
U *module directory entry pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Find Module Directory Entry call returns a pointer to the module directory entry for the first module that has a name and type matching the specified name and type. If you pass a module type of zero, the system call finds any module.
- The support module for this call is Krn.

Find 64

OS9 F$Find64 103F 2F

Returns the address of a 64-byte memory block

Entry Conditions

A *block number*
X *address of the block*

Exit Conditions

Y *address of the block*
CC *carry set if block not allowed or not in use*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- NitrOS-9 uses Find 64 to find path descriptors when given their block number. The block number can be any positive integer.

Get Free High Block

OS9 F$FreeHB 103F 3E

Searches the DAT image for the highest set of contiguous free blocks of the specified size

Entry Conditions

B *block count*
Y *DAT image pointer*

Exit Conditions

A *starting block number*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Get Free High Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is Krn.

Get Free Low Block

OS9 F$FreeLB 103F 3D

Searches the DAT image for the lowest set of contiguous free blocks of the specified size

Entry Conditions

B *block count*
Y *DAT image pointer*

Exit Conditions

A *starting block number*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Get Free Low Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is Krn.

Compact Module Directory

Compacts the entries in the module directory

OS9 F$GCMDir 103F 52

Entry Conditions

None

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This function is only for internal NitrOS-9 system use. You should never call it from a program.

Get Process Pointer

Gets a pointer to a process

OS9 F$GProcP 103F 37

Entry Conditions

A *process ID*

Exit Conditions

B *pointer to process descriptor* (if no error)

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Get Process Pointer call 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. Because of this, the address space returned only refers to system address space.
- The support module for this call is KrnP2.

I/O Delete

OS9 F$IODel 103F 33

Deletes an I/O module that is not being used

Entry Conditions

X *address of an I/O module*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The I/O Delete call deletes the specified I/O module from the system, if the module is not in use. This system call is used mainly by the I/O Manager, and can be of limited or no use for other applications.
- This is the order in which I/O Delete operates:

1. Register X passes the address of a device descriptor module, device driver module, or file manager module.
2. NitrOS-9 searches the device table for the address.
3. If NitrOS-9 finds the address, it checks the module's use count. If the count is zero, the module is not being used; NitrOS-9 deletes it. If the count is not zero, the module is being used; NitrOS-9 returns an error.

- I/O Delete returns information to the Unlink system call after determining whether a device is busy.

I/O Queue

OS9 F$IOQu 103F 2B

Inserts the calling process into another process's I/O queue, and puts the calling process to sleep

Entry Conditions

A *process ID*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The I/O Queue call links the calling process into the I/O queue of the specified process and performs an untimed sleep. The I/O Manager and the file managers are primary and extensive users of I/O Queue.
- Routines associated with the specified process send a wake-up signal to the calling process.

Set IRQ

OS9 F$IRQ 103F 2A

Adds a device to or removes it from the polling table

Entry Conditions

D *address of the device status register*
X 0 (to remove a device) or *the address of a packet* (to add a device)
- the address at X is the flip byte
- the address at X + 1 is the mask byte
- the address at X + 2 is the priority byte

Y *address of the device IRQ service routine*
U *address of the service routine's memory area*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Set IRQ is used mainly by device driver routines. (See "Interrupt Processing" in Chapter 2 for a complete discussion of the interrupt polling system.)
- Packet Definitions:
  The Flip Byte. Determines whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).
  The Mask Byte. Selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.
  The Priority Byte. Contains the device priority number (0 = lowest priority, 255 = highest priority).

Load A From Task B

Loads A from 0,X in task B

OS9 F$LDABX 103F 49

Entry Conditions

B *task number*
X *pointer to data*

Exit Conditions

A *byte at 0,X in task address space*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The value in Register X is an offset value from the beginning address of the Task module. The Load A from Task B call returns one byte from this logical address. Use this system call to get one byte from the current process's memory in a system state routine.

Get One Byte

Loads A from [X,[Y]]

OS9 F$LDAXY 103F 46

Entry Conditions

X *block offset*
Y *DAT image pointer*

Exit Conditions

A *contents of byte at DAT image (Y) offset X*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Get One Byte system call gets the contents of one byte in the specified memory block. The block is specified by the DAT image in (Y), offset by (X). The call assumes that the DAT image pointer is to the actual block desired, and that X is only an offset within the DAT block. The value in Register X must be less than the size of the DAT block. NitrOS-9 does not check to see if X is out of range.

Get Two Bytes

Load D from [D+X],[Y]

OS9 F$LDDDXY 103F 48

Entry Conditions

D *offset to the offset within the DAT image*
X *offset within the DAT image*
Y *DAT image pointer*

Exit Conditions

D *contents of two bytes at [D+X],[Y]*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Get Two Bytes loads two bytes from the address space described by the DAT image pointer. If the DAT image pointer is to the entire DAT, make D+X equal to the process address for data. If the DAT image is not the entire image (64K), you must adjust D+X relative to the beginning of the DAT image. Using D+X lets you keep a local pointer within a block, and also lets you point to an offset within the DAT image that points to a specified block number.

Map Specific Block

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

OS9 F$MapBlk 103F 4F

Entry Conditions

X *starting block number*
B *number of blocks*

Exit Conditions

U *address of first block*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The system maps blocks from the top down. It maps new blocks into the highest available addresses in the address space. See Clear Specified Block for information on unmapping.

Move Data

OS9 F$Move 103F 38

Moves data bytes from one address space to another

Entry Conditions

A *source task number*
B *destination task number*
X *source pointer*
Y *byte count*
U *destination pointer*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- You can use the Move Data system call to move data bytes from one address space to another, usually from system to user, or vice versa.
- The support module for this call is Krn.

Next Process

OS9 F$NProc 103F 2D

Executes the next process in the active process queue

Entry Conditions

None

Exit Conditions

Control does not return to caller.

Additional Information

- The Next Process system call takes the next process out of the active process queue and initiates its execution. If the queue contains no process, NitrOS-9 waits for an interrupt and then checks the queue again.
- The process calling Next Process must already be in one of the three process queues. If it is not, it becomes unknown to the system even though the process descriptor still exists and can be displayed by the PROCS command.

Release a Task

OS9 F$RelTsk 103F 43

Releases a specified DAT task number from use by a process, making the task's DAT hardware available for use by another task

Entry Conditions

B *task number*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The support module for this call is Krn.

Reserve Task Number

Reserves a DAT task number

OS9 F$ResTsk 103F 42

Entry Conditions

None

Exit Conditions

B *task number* (if no error)

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Reserve Task Number call finds a free DAT task number, reserves it, and returns the task number to the caller. The caller often then assigns the task number to a process.
- The support module for this call is Krn.

Return 64

Deallocates a 64-byte block of memory

OS9 F$Ret64 103F 31

Entry Conditions

A *block number*
X *address of the base page*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- See the Allocate 64 system call for more information.

Set Process DAT Image

Copies all or part of the DAT image into a process descriptor

OS9 F$SetImg 103F 3C

Entry Conditions

A *starting image block number*
B *block count*
X *process descriptor pointer*
U *new image pointer*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- While copying part or all of the DAT image, this system call also sets an image change flag in the process descriptor. This flag guarantees that as a process returns from the system. The call updates the hardware to match the new process DAT image.
- The support module for this call is Krn.

Set Process Task DAT Registers

Writes to the hardware DAT registers

OS9 F$SetTsk 103F 41

Entry Conditions

X *pointer to process descriptor*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This system call sets the process task hardware DAT registers, and clears the image change flag in the process descriptor. It also writes to DAT RAM the process's segment address information.
- The support module for this call is Krn.

System Link

OS9 F$SLink 103F 34

Adds a module from outside the current address space into the current address space

Entry Conditions

A *module type*
X *module name string pointer*
Y *name string DAT image pointer*

Exit Conditions

A *module type*
B *module revision* (if no error)
X *updated name string pointer*
Y *module entry point*
U *module pointer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The I/O system uses the System Link call to link into the current process's address space those modules specified by a device name in a user call. User calls such as Create File and Open Path use this system call.
- The support module for this call is Krn.

Request System Memory

Allocates a block of memory of the specified size from the top of available RAM

OS9 F$SRqMem 103F 28

Entry Conditions

D *byte count*

Exit Conditions

D *new memory size*
U *starting address of the memory area*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The Request System Memory call rounds the size request to the next page boundary.
- This call allocates memory only for system address space.

Return System Memory

Deallocates a block of contiguous pages

OS9 F$SRtMem 103F 29

Entry Conditions

D *number of bytes to return*
U *starting address of memory to return*; must point to an even page boundary

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Register U must point to an event page boundary.
- This call deallocates memory for system address space only.

Set SVC

Adds or replaces a system call

OS9 F$SSvc 103F 32

Entry Conditions

Y *address of the system call initialization table*

Exit Conditions

None

Error Output

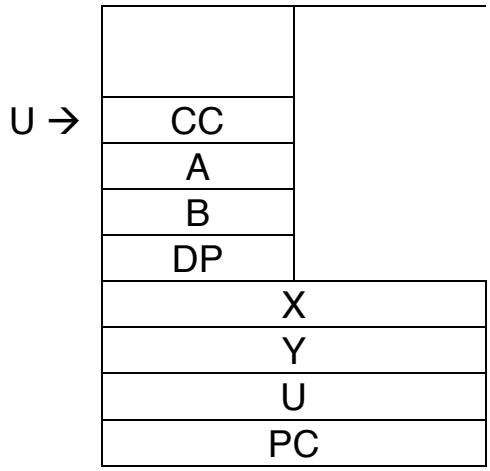
CC carry set on error
B *error code*, if any

Additional Information

- Set SVC adds or replaces a system call, which you have written, to NitrOS-9's user and system mode system call tables.
- Register Y passes the address of a table, which contains the function codes and offsets, to the corresponding system call handler routines. This table has the following format:

Relative Address	Use	
$00	Function Code	← First entry
$01 $02	Offset From Byte 3 To Function Handler	
$03	Function Code	← Second entry
$04 $05	Offset From Byte 6 To Function Handler	
	More Entries	← More entries
	$80	End-of-table mark

- If the most significant bit of the function code is set, NitrOS-9 updates the system table. If the most significant bit of the function code is not set, NitrOS-9 updates the system and user tables.
- The function request codes are in the range $29-$34. I/O calls are in the range $80-$90.
- To use a privileged system call, you must be executing a program that resides in the system map and that executes in the system state.
- The system call handler routine must process the system call and return from the subroutine with an RTS instruction.
- The handler routing might alter all CPU registers (except Register SP).
- Register U passes the address of the register stack to the system call hander as shown in the following diagram:



	Relative Address	Name
U → CC	$00	R$CC
A	$01	R$A (R$D)
B	02	R$B
DP	$03	R$DP
X	$04	R$X
Y	$06	R$Y
U	$08	R$U
PC	$0A	R$PC

Codes $70-$7F are reserved for user definition.

Store A Byte In A Task

Stores A at 0,X in Task B

OS9 F$STABX 103F 4A

Entry Conditions

A *byte to store*
B *destination task number*
X *logical destination address*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This system call is similar to the assembly language instruction “STA 0,X”. The difference is that in the system call, X refers to an address in the given task’s address space instead of the current address space.
- The support module for this system call is Krn.

Install Virtual Interrupt

OS9 F$VIRQ 103F 27

Installs a virtual interrupt handler routine

Entry Conditions

D	*initial count value*
X	0 to delete entry 1 to install entry
Y	*address of 5-byte packet*

Exit Conditions

None

Error Output

CC	carry set on error
B	*error code*, if any

Additional Information

- Install VIRQ for use with devices in the Multi-Pak Expansion Interface. This call is explained in detail in Chapter 2.

Validate Module

Checks the module header parity and CRC bytes of a module

OS9 F$VModul 103F 2E

Entry Conditions

D *DAT image pointer*
X *new module block offset*

Exit Conditions

U *address of the module directory entry*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- If the values of the specified module are valid, NitrOS-9 searches the module directory for a module with the same name. If one exists, NitrOS-9 keeps in memory the module that has the higher revision level. If both modules have the same revision level, NitrOS-9 retains the module in memory.

Get Status System Calls

You use the Get Status system calls with the RBF manager subroutine GETSTA. The NitrOS-9 Level Two system reserves the function codes 7-127 for use by Microware. You can define codes 128-255 and their parameter-passing conventions for your own use. (See the sections on device drivers in Chapters 4, 5, and 6.)

The Get Status routine passes the register stack and the specified function code to the device driver.

Following are the Get Status functions and their codes.

SS.OPT

Function Code $00

Reads the option section of the path descriptor, and passes it into the 32-byte area pointed to by Register X

Entry Conditions

A *path number*
B $00
X *address to receive status packet*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Use SS.OPT to determine the current settings for editing functions, such as echo and auto line feed.

SS.RDY

Tests for data available on SCF-supported devices

Function Code $01

Entry Conditions

A *path number*
B $01

Exit Conditions

If the device is ready:

CC carry clear
B $00

On devices that support it (both CC3IO and ACIPAK support this), Register B returns the number of characters that are ready to be read.

If the device is ready:

CC carry set
B $F6 (E$SRNDY)

Error Output

CC carry set on error
B *error code*, if any

SS.SIZ

Gets the current file size (RBF-supported devices only)

Function Code $02

Entry Conditions

A *path number*
B $02

Exit Conditions

X *most significant 16 bits of the current file size*
U *least significant 16 bytes of the current file size*

Error Output

CC carry set on error
B *error code*, if any

SS.POS

Function Code $05

Gets the current file position (RBF-supported devices only)

Entry Conditions

A *path number*
B $05

Exit Conditions

X *most significant 16 bits of the current file position*
U *least significant 16 bits of the current file position*

Error Output

CC carry set on error
B *error code*, if any

SS.EOF

Tests for the end of file (EOF)

Function Code $06

Entry Conditions

A *path number*
B $06

Exit Conditions

If there is no EOF:

CC carry clear
B $00

If there is an EOF:

CC carry set
B $D3 (E$EOF)

Error Output

CC carry set on error
B *error code*, if any

SS.DevNm

Returns a device name

Function Code $0E

Entry Conditions

A	*path number*
B	$0E
X	*address of 32-byte buffer for name*

Exit Conditions

X	*address of buffer, name moved to buffer*

Error Output

CC	carry set on error
B	*error code*, if any

SS.DSTAT

Returns the display status

Function Code $12

Entry Conditions

A *path number*
B $12

Exit Conditions

A *color code of the pixel at the cursor address*
X *address of the graphics display memory*
Y *graphics cursor address*: X = MSB, Y = LSB

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This function is supported only with the VDGINT module and deals with VDG-compatible graphics screens. See SS.AAGBf for information regarding Level Two operation.

SS.JOY

Returns the joystick values

Function Code $13

Entry Conditions

A *path number*
B $13
X *joystick number*
0 = right joystick
1 = left joystick

Exit Conditions

A *fire button down*
0 = none
1 = Button 1
2 = Button 2
3 = Button 1 and Button 2
X *selected joystick x value* (0-63)
Y *selected joystick y value* (0-63)
Note: Under Level 1, the following values are returned by this call:
A *fire button status*
$FF = fire button is on
$00 = fire button is off

Error Output

CC carry set on error
B *error code*, if any

SS.AlfaS

Function Code $1C

Returns VDG alpha screen memory information

Entry Conditions

A *path number*
B $1C

Exit Conditions

A *caps lock status*
$00 = lower case
$FF = upper case
X *memory address of the buffer*
Y *memory address of the cursor*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.AlfaS maps the screen into the user address space. The call requires a full block of memory for screen mapping. This call is only for use with VDG text screens handled by VDGINT.
- The support module for this call is VDGINT.
- **Warning**: Use extreme care when poking the screen since other system variables reside in screen memory. Do not change any addresses outside of the screen.

SS.Cursr

Function Code $25

Returns VDG alpha screen cursor information

Entry Conditions

A *path number*
B $25

Exit Conditions

A *character code of the character at the current cursor address*
X *cursor X position (column)*
Y *cursor Y position (row)*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.Cursr returns the character at the current cursor position. It also returns the X-Y address of the cursor relative to the current device's window or screen. SS.Cursr works only with text screens.
- The support module for this call is VDGINT.

SS.ScSiz

Returns the window or screen size

Function Code $26

Entry Conditions

A *path number*
B $26

Exit Conditions

X *number of columns on screen/window*
Y *number of rows on screen/window*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Use this call to determine the size of an output screen. The values returned depend on the device in use:
  - For non-CCIO devices, the call returns the values following the XON/XOFF bytes in the device descriptor.
  - For CCIO devices, the call returns the size of the window or screen in use by the specified device.
  - For window devices, the call returns the size of the current working area of the window.
- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.KySns

Returns key down status

Function Code $27

Entry Conditions

A *path number*
B $27

Exit Conditions

A *keyboard scan information*

Additional Information

- Accumulator A returns with a bit pattern representing eight keys. With each keyboard scan, NitrOS-9 updates this bit pattern. A set bit (1) indicates that a key was pressed since the last scan. A clear bit (0) indicates that a key was not pressed. Definitions for the bits are as follows:

Bit	Key
0	SHIFT
1	CTRL or CLEAR
2	ALT or @
3	Up arrow
4	Down arrow
5	Left arrow
6	Right arrow
7	Space Bar

  The bits can be makes with the following equates:

```
SHIFTBIT  equ  %00000001
CNTRLBIT  equ  %00000010
ALTERBIT  equ  %00000100
UPBIT     equ  %00001000
DOWNBIT   equ  %00010000
LEFTBIT   equ  %00100000
RIGHTBT   equ  %01000000
SPACEBIT  equ  %10000000
```

- The support module for this call is CC3IO.

SS.ComSt

Function Code $28

Returns serial port configuration information

Entry Conditions

A *path number*
B $28

Exit Conditions

Y *high byte: parity*
low byte: baud
(See the SetStat call SS.ComSt for values)

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The SCF manager uses this call when performing an SS.Opt GetStat on an SCF-type device. Use calls to SS.ComSt do not update the path descriptor. Use the SS.OPT GetStat call for most applications because it automatically updates the path descriptor.

SS.MnSel

Function Code $87

Requests that the high-level menu handler take control of menu selection

Entry Conditions

A *path number*
B $87

Exit Conditions

A *Menu ID* (if valid selection)
0 (if invalid selection)
B *item number of menu (if valid selection)*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- After detecting a valid a valid mouse click (when the mouse is pointing to a control area on a window), a process needs to call SS.MnSel. This call tells the enhanced window interface to handle any menu selection being made. If accumulator A returns with 0, no selection has been made. The calling process needs to test and handle other returned values.
- A condition where Register A returns a valid menu ID number and Register B returns 0 signals the selection of a menu with no items. The application can now take over and do a special graphics pull down of its own. The menu title remains highlighted until the application calls the SS.UMBar SetStat to update the menu bar.
- The support module for this call is WindInt.

SS.Mouse

Gets mouse status

Function Code $89

Entry Conditions

A *path number*
B $89
X *data storage area address*
Y *mouse port select:*
0 = automatic selection
1 = right side
2 = left side

Exit Conditions

X *data storage area address*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.Mouse returns information on the current mouse and its fire button. The following list defines the 32-byte data packet that SS.Mouse creates:

Pt.Valid	rmb	1	Is returned info valid? (0 = no, 1 = yes)	
Pt.Actv	rmb	1	Active side (0 = off, 1 = right, 2 = left)	
Pt.ToTm	rmb	1	Timeout initial value	
	rmb	2	Reserved	
Pt.TTTo	rmb	1	Time until timeout	
	rmb	2	Reserved	
Pt.TSSt	rmb	2	Time since counter start	
Pt.CBSA	rmb	1	Current button state	(Button A)
Pt.CBSB	rmb	1	Current button state	(Button B)
Pt.CCtA	rmb	1	Click count	(Button A)
Pt.CCtB	rmb	1	Click count	(Button B)
Pt.TTSA	rmb	1	Time this state counter	(Button A)
Pt.TTSB	rmb	1	Time this state counter	(Button B)
Pt.TLSA	rmb	1	Time last state counter	(Button A)
Pt.TLSB	rmb	1	Time last state counter	(Button B)
	rmb	2	Reserved	
Pt.BDX	rmb	2	Button down frozen	Actual X
Pt.BDY	rmb	2	Button down frozen	Actual Y
Pt.Stat	rmb	1	Window pointer type location	
Pt.Res	rmb	1	Resolution (0-640 by 0=10/1=1)	
Pt.AcX	rmb	2	Actual X value	
Pt.AcY	rmb	2	Actual Y value	
Pt.WRX	rmb	2	Window relative X	
Pt.WRY	rmb	2	Window relative Y	
Pt.Siz	equ	.	Packet size 32 bytes	

- Button Information:

Pt.Valid. The valid byte gives the caller an indication of whether the information contained in the returned packet is accurate. The information returned by this call is only valid if the process is running on the current interactive window. If the process is on a non-interactive window, the byte is zero and the process can ignore the information returned.

Pt.Actv. This byte shows which port is selected for use by all mouse functions. The default value is Right (1). You can change this value with the SS.GIP SetStat call.

Pt.ToTm. This is the initial value used by Pt.TTTo.

Pt.TTTo. This is the count down value (as of the instant the GetStat call is made). This value starts at the value contained in the Pt.ToTim and counts down to zero at a rate of 60 counts per second. The system maintains all counters until this value reaches 0, at which point it sets all counters and states to 0. The mouse scan routine changes into a quiet mode, which requires less overhead than when the mouse is active. The timeout begins when both buttons are in the up (open) state. The timer is reinitialized to the value in Pt.ToTm when either button goes down (closed).

Pt.TSSt. This counter is constantly increasing, beginning when either button is pressed while the mouse is in the quiet state. All counts are a number of ticks (60 per second). The timer counts to $FFFF, then stays at that value if additional ticks occur.

Pt.CBSA/Pt.CBSB. These flag bytes indicate the state of the button at the last system clock tick. A value of 0 indicates that the button is up (open). A value other than zero indicates that the button is down (closed). Button A is available on all Tandy joysticks and mice. Button B is only available for products that have two buttons.

The system scans the mouse buttons each time it scans the keyboard.

Pt.CCtA/Pt.CCtB. This is the number of clicks that have occurred since the mouse went into an active state. A click is defined as pressing (closing) the button, then releasing (opening) the button. The system counts the clicks as the button is released.

Pt.TTSA/Pt.TTSB. This counter is the number of ticks that have occurred during the current state, as defined by Pt.CBSx. This counter starts at one (counts the tick when the state changes) and increases by one for each tick that occurs while the button remains in the same state (open or closed).

Pt.TSLA/Pt.TLSB. This counter is the number of ticks that have occurred during the time that a button is in a state opposite of the current state. Using this count and the TTSA/TTSB count, you can determine how a button was in the previous state, even if the system returns the packet after

the state has changed. Use these counters, along with the state and click count values, to define any type of click, drag, or hold convention you want.

Reserved. Two packet bytes are reserved for future expansion of the returned data.

- Position Information:

Pt.BDX/Pt.BDY. These values are copies of the Pt.AcX and Pt.AcY values when either of the buttons change from an open state to a closed state.

Pt.Stat. This byte contains information about the area of the screen on which the mouse is positioned. Pt.Valid must be a value other than 0 for this information to be accurate. If Pt.Valid is 0, this value is also 0 and not accurate. Three types of areas are currently defined:

0 = content region or current working area of the window
1 = control region (for use by Multi-View)
2 = off window, or on an area of the screen that is not part of the window.

Pt.Res. This value is the current resolution for the mouse. The mouse must always return coordinates in the range of 0-639 for the X axis and 0-191 for the Y axis. If the system is so configured, you can use the high-resolution mouse adapter that provides a 1-to-1 ratio with these values plus 1. If the adapter is not in use, the resolution is a ration of 1-to-10 on the X axis and 1-to-3 on the Y axis. The keyboard mouse provides a resolution of 1-to-1. The values in Pt.Res are:

0 = low res (x:10, y:3)
1 = high res (x,y:1)

Pt.AcX/PT.AcY. the values read from the mouse returned in the resolution as described under Pt.Res.

Pt.WRX/Pt.WRY. The values read from the mouse minus the starting coordinates of the current window's working area. These values return the coordinates in numbers relative to the type of screen. For example, the X axis is in the range 0-639 for high-resolution screens and 0-319 for low resolution screens. You can divide the window relative values by 8 to obtain absolute character positions. These values are most helpful when working in non-scaled modes.

- The support modules for this call are CC3IO, GrfInt, and WindInt.

SS.Palet

Gets palette information

Function Code $91

Entry Conditions

A *path number*
B $91
X *pointer to the 16-byte palette information buffer*

Exit Conditions

X *pointer to the 16-byte palette information buffer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.Palet reads the contents of the 16 screen palette registers, and stores them in a 16-byte buffer. When you make the call, be sure the X register points to the desired buffer location. The pointer is retained on exit. The palette values returned are specific to the screen on which the call is made.
- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.ScTyp

Function Code $93

Returns the type of a screen to a calling program

Entry Conditions

A *path number*
B $93

Exit Conditions

A *screen type code*
1 = 40 x 24 text screen
2 = 80 x 24 text screen
3 = not used
4 = not used
5 = 640 x 192, 2-color graphics screen
6 = 320 x 192, 4-color graphics screen
7 = 640 x 192, 4-color graphics screen
8 = 320 x 192, 16-color graphics screen

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Support modules for this system call are GrfInt and WindInt.

SS.FBRgs

Function Code $96

Returns the foreground, background, and border palette registers for a window

Entry Conditions

A *path number*
B $96

Exit Conditions

A *foreground palette register number*
B *background palette register number* (if carry clear)
X *least significant byte of border palette register number*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Support modules for SS.FBRgs are GrfInt and WindInt.

SS.DFPal

Returns the default palette register settings

Function Code $97

Entry Conditions

A *path number*
B $97
X *pointer to 16-byte data space*

Exit Conditions

X *default palette data moved to use space*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- You can use SS.DFPal to find the values of the default palette registers that are used when a new screen is allocated by GrfInt or WindInt. The corresponding SetStat can alter the default registers. This GetStat/SetStat pair is for system configuration utilities and should not be used by general applications.

Set Status System Calls

Use the Set Status system calls with the RBF manager subroutine SETSTA. The NitrOS-9 Level Two system reserves function codes 7-127 for use by Microware. You can define codes 200-255 and their parameter-passing conventions for your own use. (See the section on device drivers in Chapters 4, 5, and 6.)

Following are the Set Status functions and their codes.

SS.OPT

Function Code $00

Writes the option section of the path descriptor

Entry Conditions

A *path number*
B $00

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.OPT writes the option section of the path descriptor from the 32-byte status packet pointed to by Register X. Use this system call to set the device operating parameters, such as echo and line feed.

SS.SIZ

Changes the size of a file for RBF-type devices

Function Code $02

Entry Conditions

A *path number*
B $02
X *most significant 16 bits of the desired file size*
U *least significant 16 bits of the desired file size*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

SS.RESET

Function Code $03

Restores the disk drive head to Track 0 in preparation for formatting and error recovery (use only with RBF-type devices)

Entry Conditions

A *path number*
B $03

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

SS.WTRK

Function Code $04

Formats (writes) a track on a disk (RBF-type devices only)

Entry Conditions

A *path number*
B $04
U *track number* (least significant 8 bits)
X *address of the track buffer*
Y *side/density*
Bit 0 = side
0 = side 0
1 = side 1
Bit 1 = density
0 = single
1 = double

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- For hard disks or floppy disks that have a “format entire diskette” command, SS.WTRK formats the entire disk only when the *track number* is zero.

SS.SQD

Function Code $0C

Starts the shutdown procedure for a hard disk that has sequence-down requirements prior to removal of power. (Use only with RBF-type devices.)

Entry Conditions

A *path number*
B $0C

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

SS.KySns

Turns the key sense function on and off

Function Code $27

Entry Conditions

A *path number*
B $27
X *key sense switch value*
0 = normal key operation
1 = key sense operation

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- When SS.KySns switches the keyboard to key sense mode, the CC3IO module suspends transmission of keyboard characters to the SCF manager and the user. While the computer is in key sense mode, the only way to detect key press is with SS.KySns.
- The support module for this call is CC3IO.

SS.ComSt

Function Code $28

Used by the SCF manager to configure a serial port

Entry Conditions

A *path number*
B $28
Y *high byte: parity*
low byte: baud rate

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- **Baud Configuration**. The high order byte of Y determines the baud rate, the word length, and the number of stop bits. The byte is configured as follows:

  Bit 0-3 Baud rate
  Bit 4 reserved
  Bit 5-6 Word length
  Bit 7 Stop bits

  Stop bits:
  0 = 1
  1 = 2

  Word length:
  00 = 8 bit
  01 = 7 bit

  Baud rate:
  0000 = 110
  0001 = 300
  0010 = 600

0011 = 1200
0100 = 2400
0101 = 4800
0110 = 9600
0111 = 19,200
1xxx = undefined

- **Parity Configuration**. The low order byte of Y determines parity. The byte is configured as follows:

  Bits 0-4 Special use
  Bits 5-7 Parity

  Parity:
  xx0 = none
  001 = odd (ACIAPAK and MODPAK only)
  011 = even (ACIAPAK and MODPAK only)
  101 = transmit: mark
  receive: ignore
  111 = transmit: space
  receive: ignore

- The SCF manager uses SS.ComSt to inform a driver that serial port configuration information has been changed in the path descriptor. After calling SS.ComSt, a user routine must call the SS.OPT SetStat to correctly update the path descriptor.
- This call is for the use of the SCF manager. Use SS.OPT for changing baud, stop bit, and parity values.

SS.Close

Function Code $2A

Informs a device driver that a path is closed

Entry Conditions

A *path number*
B $2A

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- This call is used internally by NitrOS-9's SCF file manager and is not available for user programs. It can be used only by device drivers and file managers.

SS.AAGBf

Reserves an additional graphics buffer

Function Code $80

Entry Conditions

A *path number*
B $80

Exit Conditions

X *buffer address*
Y *buffer number (1-2)*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.AAGBf allocates an additional 8K graphics buffer. The first buffer (Buffer 0) must be allocated by using the Display Graphics command. To use the Display Graphics command, send control code $0F to the standard terminal driver. SS.AAGBf can allocate up to two additional buffers (Buffers 1 and 2), one at a time.
- After calling SS.AAGBf, Register X contains the address of the new buffer. Register Y contains the buffer number.
- To deallocate all graphics buffers, use the End Graphics control code.
- When SS.AAGBf allocates a buffer, it also maps the buffer into the application's address space. Each buffer uses 8K of the available memory in the application's address space. Also, if SS.DStat is called, Buffer 0 is also mapped into the application's address space. Allocation of all three buffers reduces the application's free memory by 24K.
- The support module for this call is VDGINT.

SS.SLGBf

Selects a graphics buffer

Function Code $81

Entry Conditions

A	*path number*	
B	$81	
X	$00	select buffer for use
	$01-$FF	select buffer for use and display
Y	*buffer number* (0-2)	

Exit Conditions

X	*unchanged from entry*
Y	*unchanged from entry*

Error Output

CC	carry set on error
B	*error code*, if any

Additional Information

- Use Display Graphics to allocate the first graphics buffer. Use SS.AAGBf to allocate the second and third graphics buffers.
- Save each return address when writing directly to a screen. It is not necessary to save return addresses when using operating system graphics commands.
- SS.SLBGf does not update hardware information until the next vertical retrace (60Hz rate). Programs that use SS.AAGBf to change current draw buffers need to wait long enough to ensure that NitrOS-9 has moved the current buffer to the screen.
- The screen shows the buffer only if the buffer is selected as the interactive device. If the device does not possess the keyboard, NitrOS-9 stores the information until the device is selected as the interactive device. When the device is selected as the interactive device, the display shows the selected device's screen.
- The support module for this call is VDGINT.

SS.MpGPB

Function Code $84

Maps the Get/Put buffer into a user address space

Entry Conditions

A *path number*
B $84
X *high byte: buffer group number*
low byte: buffer number
Y *action to take*:
1 = map buffer
0 = unmap buffer

Exit Conditions

X *address of the mapped buffer*
Y *number of bytes in buffer*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.MpGPB maps a Get/Put buffer into the user address space. You can then save the buffer to disk or directly modify the pixel data contained in the buffer. Use extreme care when modifying the buffer so that you do not write outside of the buffer data area.

SS.WnSet

Set up a high level window handler

Function Code $86

Entry Conditions

A *path number*
B $86
X *window data pointer* (if Y=WT.FSWin or WT.Win)
Y *window type code*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- The C language data structures for windowing are defined in the wind.h file in the DEFS directory of the system disk.
- The support module for this call is WindInt.

SS.SBar

Puts a scroll block at a specified position

Function Code $88

Entry Conditions

A *path number*
B $88
X *horizontal position of scroll block*
Y *vertical position of scroll block*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- WT.FSWin-type windows have areas at the bottom and right sides to indicate their relative positions within a larger area. These areas are called scroll bars. SS.SBar gives an application the ability to maintain relative position markers within the scroll bars. The markers indicate the location of the current screen within a larger screen. Calling SS.SBar updates both scroll markers.
- The support module for this call is WindInt.

SS.MsSig

Function Code $8A

Sends a signal to a process when the mouse button is pressed

Entry Conditions

A *path number*
B $8A
X *user defined signal code* (low byte only)

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.MsSig sends the process a signal the next time a mouse button changes state (from open to closed). Once SS.MsSig sends the signal, the process must repeat the SetStat each time that it needs to set up the signal.
- Processes using SS.MsSig should have an intercept routine to trap the signal. By intercepting the signal, other processes can be notified when the change occurs. Therefore, the other processes do not need to continually poll the mouse.
- The SS.Relea SetStat clears the pending signal request, if desired. It also clears any pending signal from SS.SSig. Because of this, if you want to clear only one signal, you must reset the other signal after calling SS.MsSig.
- The support module for this call is CC3IO.

SS.AScrn

Function Code $8B

Allocates and maps a high-resolution screen into an application address space

Entry Conditions

A *path number*
B $8B
X *screen type*
0 = 640 x 192 x 2 colors (16K)
1 = 320 x 192 x 4 colors (16K)
2 = 160 x 192 x 16 colors (16K)
3 = 640 x 192 x 4 colors (32K)
4 = 320 x 192 x 16 colors (32K)

Exit Conditions

X *application address space of screen*
Y *screen number (1-3)*

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.AScrn is particularly useful in systems with minimal memory when you want to allocate a high resolution graphics screen with all screen updating handled by a process.
- The call uses VDGInt (GRFINT is not required).
- All screens are allocated in multiples of 8K blocks. You can allocate a maximum of three buffers at one time. To select between buffers, use the SS.DScrn SetStat call.
- Screen memory is allocated but not cleared. The application using the screen must do this.
- Screens must be allocated from a VDG-type device—a standard 32-column text screen must be available for the device.
- The support module for this call is VDGINT.

SS.DScrn

Causes VDGINT to display a screen that was allocated by SS.AScrn

Function Code $8C

Entry Conditions

A *path number*
B $8C
Y *screen number*
0 = text screen (32 x 16)
1-3 = high resolution screen number

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.DScrn shows the requested screen if the requested screen is the current interactive device.
- Screen 0 (text screen) should be selected before using SS.FScrn to free all high resolution screen memory.
- The support module for this call is VDGINT.

SS.FScrn

Function Code $8D

Frees the memory of a screen allocated by SS.AScrn

Entry Conditions

A *path number*
B $8D
Y *screen number (1-3)*

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Do not attempt to free a screen that is currently on the display.
- SS.FScrn returns the screen memory to the system and removes it from an application's address space.
- The support module for this call is VDGINT.

SS.PScrn

Converts a screen to a different type

Function Code $8E

Entry Conditions

A *path number*
B $8E
X *new screen type*
0 = 640 x 192 x 2 colors (16K)
1 = 320 x 192 x 4 colors (16K)
2 = 160 x 192 x 16 colors (16K)
3 = 640 x 192 x 4 colors (32K)
4 = 320 x 192 x 16 colors (32K)

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.PScrn changes a screen allocated by SS.AScrn to a new screen type. You can change a 32K screen to either a 32K screen or a 16K screen. You can change a 16K screen only to another 16K screen type. SS.PScrn updates the current display screen at the next clock interrupt.
- If you change a 32K screen to a 16K screen, NitrOS-9 does not reclaim the extra 16K of memory. This means that you can later change the 16K screen back to a 32K screen.
- The support module for this call is VDGINT.

SS.Montr

Sets the monitor type

Function Code $92

Entry Conditions

A *path number*
B $92
X *monitor type*
0 = color composite
1 = analog RGB
2 = monochrome composite

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- SS.Montr loads the hardware palette registers with the codes for the default color set for three types of monitors. The system default initializes the palette for a composite color monitor.
- The monochrome mode removes color information from the signals sent to a monitor.
- When a composite monitor is in use, a conversion table maps colors from RGB color numbers. In RGB and monochrome modes, the system uses the RGB color numbers directly.
- The support modules for this call are VDGINT and GrfDrv.

SS.GIP

Function Code $94

Sets the system wide mouse and key repeat parameters

Entry Conditions

A *path number*
B $94
X *mouse resolution*; in the most significant byte
0 = low resolution mouse
1 = optional high resolution adapter
Mouse port location; in the least significant byte
1 = right port
2 = left port
Y *key repeat start constant*; in the most significant byte
key repeat delay; in the least significant byte
00XX = no repeat
FFFF = unchanged

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Because this function affects system-wide settings, it is best to use it from system configuration utilities and not from general application programs.
- The support module for this call is CC3IO.

SS.UMBar

Function Code $95

Requests the high level menu manager to update the menu bar

Entry Conditions

A *path number*
B $95

Exit Conditions

None

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- An application can call SS.UMBar when it needs to redraw menu bar information, such as when it enables or disables menus, or when it completes a window *pull down* and needs to restore the menu.
- The support module for this call is WindInt.

SS.DFPal

Sets the default palette register values

Function Code $97

Entry Conditions

A *path number*
B $97
X *pointer to 16 bytes of palette data*

Exit Conditions

X unchanged, bytes moved to system defaults

Error Output

CC carry set on error
B *error code*, if any

Additional Information

- Use SS.DFPal to alter the system-wide palette register defaults. The system uses these defaults when it allocates a new screen using the DWSet command.
- Because this function affects system wide settings, it is best to use it from system configuration utilities, not general application programs.

SS.Tone

Creates a sound through the terminal output device

Function Code $98

Entry Conditions

A *path number*
B $98
X *duration and amplitude of the tone*
LSB = duration in ticks (60-sec) in the range 0-255
MSB = amplitude of tone in the range 0-63
Y *relative frequency counter* (0=low, 4095=high)

Exit Conditions

These are the same as the entry conditions.

Error Output

There are no error conditions.

Additional Information

- This call produces a programmed I/O tone through the speaker of the monitor used by the terminal device. You can make the call on any valid path open to term to a window device.
- The system does not mask interrupts during the time the tone is being produced.
- The frequency of the tone is a relative number ranging from 0 for a low frequency to 4095 for a high frequency. The widest variation of tones occurs at the high range of the scale.

Chapter 9. Appendices

A. System Module Diagrams

B. Standard Floppy Disk Format

Color Computer 3

Physical Track Format Pattern

Format	Bytes (Dec)	Value (Hex)
Header pattern (once per track)	32	4E
	12	00
	3	F5
	1	FC
	32	4E
Sector pattern (repeated 18 times)	12	00
	3	F5
	1	Track number (0-34)
	1	Side number (0-1)
	1	Sector number (1-18)
	1	Sector length code (1)
	2	CRC
	22	4E
	12	00
	3	F5
	1	FB
	256	Data area
	2	CRC
	24	4E
Trailer pattern (once per track)	N	4E (fill to index mark)

C. System Error Codes

The error codes are show in both hexadecimal and decimal. The error codes listed include NitrOS-9 system error codes, BASIC error codes, and standard windowing system error codes.

Code		Code Meaning
HEX	**DEC**	
$01	001	UNCONDITIONAL ABORT—An error occurred from which NitrOS-9 cannot recover. All processes are terminated.
$02	002	KEYBOARD ABORT—You pressed BREAK to terminate the current operation.
$03	003	KEYBOARD INTERRUPT—You pressed SHIFT-BREAK either to cause the current operation to function as a background task with no video display or to cause the current task to terminate.
$B7	183	ILLEGAL WINDOW TYPE—You tried to define a text type window for graphics or used illegal parameters.
$B8	184	WINDOW ALREADY DEFINED—You tried to create a window that is already established.
$B9	185	FONT NOT FOUND—You tried to use a window font that does not exist.
$BA	186	STACK OVERFLOW—Your process (or processes) requires more stack space than is available on the system.
$BB	187	ILLEGAL ARGUMENT—You have used an argument with a command that is inappropriate.
$BD	189	ILLEGAL COORDINATES—You have given coordinates to a graphics command that are outside the screen boundaries.
$BE	190	INTERNAL INTEGRITY CHECK—System modules or data are changed and are no longer reliable.
$BF	191	BUFFER SIZE TOO SMALL—The data you assigned to a buffer is larger than the buffer.
$C0	192	ILLEGAL COMMAND—You have issued a command in a form unacceptable to NitrOS-9.
$C1	193	SCREEN OR WINDOW TABLE IS FULL—You do not have enough room in the system window table to keep track of any more windows or screens.
$C2	194	BAD/UNDEFINED BUFFER NUMBER—You have specified an illegal or undefined buffer number.
$C3	195	ILLEGAL WINDOW DEFINITION—You have tried to give a window

Code		Code Meaning
HEX	DEC	
		illegal parameters.
$C4	196	WINDOW UNDEFINED—You have tried to access a window that you have not yet defined.
$C8	200	PATH TABLE FULL—NitrOS-9 cannot open the file because the system path table is full.
$C9	201	ILLEGAL PATH NUMBER—The path number is too large or you specified a non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL—Your system cannot handle an interrupt request because the polling table does not have room for more entries.
$CB	203	ILLEGAL MODE—The specified device cannot perform the indicated input or output function.
$CC	204	DEVICE TABLE FULL—The device table does not have enough room for another device.
$CD	205	ILLEGAL MODULE HEADER—NitrOS-9 cannot load the specified module because its sync code, header parity, or Cyclic Redundancy Code is *incorrect*.
$CE	206	MODULE DIRECTORY FULL—The module directory does not have enough room for another module entry.
$CF	207	MEMORY FULL—Process address space is full or your computer does not have sufficient memory to perform the specified task.
$D0	208	ILLEGAL SERVICE REQUEST—The current program has issued a system call containing an illegal code number.
$D1	209	MODULE BUSY—Another process is already using a non-shareable module.
$D2	210	BOUNDARY ERROR—NitrOS-9 has received a memory allocation or deallocation request that is not on a page boundary.
$D3	211	END OF FILE—A read operation has encountered an end-of-file character and has terminated.
$D4	212	RETURNING NON-ALLOCATED MEMORY—The current operation has attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT—The file structure of the specified device is damaged.
$D6	214	NO PERMISSION—The attributes of the specified file or device do not permit the requested access.
$D7	215	BAD PATHNAME—The specified pathlist contains a syntax error; for instance, an illegal character.

Code		Code Meaning
HEX	DEC	
$D8	216	PATH NAME NOT FOUND—The system cannot find the specified pathlist.
$D9	217	SEGMENT LIST FULL—The specified file is too fragmented for further expansion.
$DA	218	FILE ALREADY EXISTS—The specified filename already exists in the specified directory.
$DB	219	ILLEGAL BLOCK ADDRESS—The file structure of the specified device is damaged.
$DC	220	PHONE HANGUP-DATA CARRIER LOST—The data carrier detect is lost on the RS-232 port.
$DD	221	MODULE NOT FOUND—The system received a request to link a module that is not in the specified directory.
$DE	222	SECTOR OUT OF RANGE—A disk sector number was specified that does not exist.
$DF	223	SUICIDE ATTEMPT—The current operation has attempted to return to the memory location of the stack.
$E0	224	ILLEGAL PROCESS NUMBER—The specified process does not exist.
$E2	226	NO CHILDREN—The system has issued a *wait service* request but the current process has no dependent process to execute.
$E3	227	ILLEGAL SWI CODE—The system received a software interrupt code that is less than 1 or greater than 3.
$E4	228	PROCESS ABORTED—The system received a signal Code 2 to terminate the current process.
$E5	229	PROCESS TABLE FULL—A fork request cannot execute because the process table has no room for more entries.
$E6	230	ILLEGAL PARAMETER AREA—A fork call has passed incorrect high and low bounds.
$E7	231	KNOWN MODULE—The specified module is for internal use only.
$E8	232	INCORRECT MODULE CRC—The CRC for the module being accessed is bad.
$E9	233	SIGNAL ERROR—The receiving process has a previous, unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE—The system cannot locate the specified module.
$EB	235	BAD NAME—The specified device, file, or module name is illegal.
$EC	236	BAD HEADER—The specified module header parity is incorrect.

Code		Code Meaning
HEX	DEC	
$ED	237	RAM FULL—No free system random access memory is available: the system address space is full, or there is no physical memory available when requested by the operating system in the system state.
$EE	238	UNKNOWN PROCESS ID—The specified process ID number is incorrect.
$EF	239	NO TASK NUMBER AVAILABLE—All available task numbers are in use.

Device Driver Errors

I/O device drivers generate the following error codes. In most cases, the codes are hardware-dependent. Consult your device manual for more details.

Code		Code Meaning
HEX	DEC	
$F0	240	UNIT ERROR—The specified device unit does not exist.
$F1	241	SECTOR ERROR—The specified sector number is out of range.
$F2	242	WRITE PROTECT—The specified device is write-protected.
$F3	243	CRC ERROR—A Cyclic Redundancy Code error occurred on a read or write verify.
$F4	244	READ ERROR—A data transfer error occurred during a disk read operation, or there is a SCN (terminal) input buffer overrun.
$F5	245	WRITE ERROR—An error occurred during a write operation.
$F6	246	NOT READY—The device specified has a *not ready* status.
$F7	247	SEEK ERROR—The system attempted a seek operation on a non-existent sector.
$F8	248	MEDIA FULL—The specified media has insufficient free space for the operation.
$F9	249	WRONG TYPE—An attempt is made to read incompatible media (for instance an attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY—A non-shareable device is in use.
$FB	251	DISK ID CHANGE—You changed diskettes when one or more files are open.
$FC	252	RECORD IS LOCKED-OUT—Another process is accessing the requested record.

Code		Code Meaning
HEX	**DEC**	
$FD	253	NON-SHAREABLE FILE BUSY—Another process is accessing the requested file.
$FE	254	I/O DEADLOCK—Two processes have attempted to gain control of the same disk area at the same time.

Index